Comparison of network complexity measures

Yipei Zhao

A thesis presented for the degree of
Master of Science Data Analytics
Supervised by Jens Christian Claussen
08 October 2021

Abstract

The goal of this project is to understand the differences and uniqueness of complexity measures. Understanding complexity measures will help the community to build better measures that are applicable and comprehensive. We implemented seven introduced measures and a new measure MA_{RI} to monitor their behaviours on graph models and real networks. MA_{RI} is a well-behaved and computationally cheap product measure that will assign the highest complexity to graphs with briefly less number of edges than medium. We investigated the key difference between random networks and real networks with a special type of network: transportation networks. Transportation networks have extremely high average distances and lower complexity than other real networks due to the general nature of transportation networks.

Contents

1 Introduction

In the modern world, there are many fields where a network can be investigated and studied. For example, electrics in a region would form a large network.[1] Also, analysing social networks and use their unique properties to deliver promotion precisely would help firms to increase profits.[2] In addition, food webs are classic networks that have been studied for many years.[3] There are many aspects of a network, but it is vital to ask: what is the complexity of a network? How to design a simple/complex network? The goal of this project is to summarise and compare different complexity measures and demonstrate their properties and uniqueness, they are implemented and used on real networks.

In this project, we want to be specific on transport networks. A transport network transfers either passengers or goods, from an initial location to their destination. The designers have to examine the complexities of the designed networks, to ensure they meet the requirement of the demand. If they don't, how can the networks be modified to reach their corresponding demand? Transport networks are different to other networks in the real world, but before investigating the uniqueness of transport networks, it is vital to study network science in general.
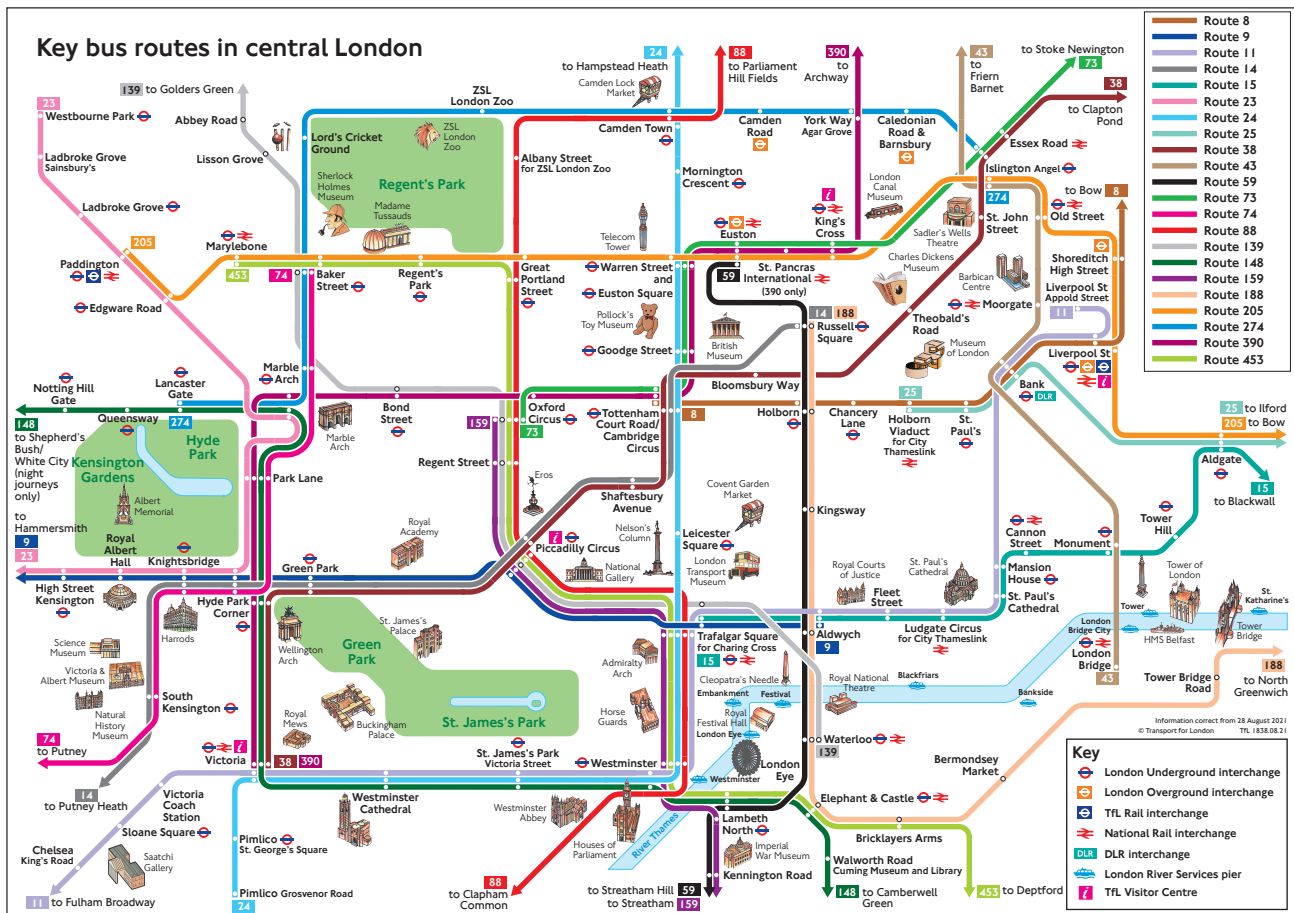

Figure 1: Key routes in central London.[4]

1.1   Graph theory

Firstly, what do network scientists study? A network contains n nodes/vertices and m edges/links; nodes can be connected using edges. Nodes and edges allow a relationship to be constructed between assets. In a transport network, a node can be considered as a stop on the road, edges are the roads/path connecting stops.

"we view network science as the study of the collection, management, analysis, interpretation, and presentation of relational data" [5]

1.1.1   Graph

To stay simple, all networks in this project have the following properties:

- Connected. All graphs we are using are connected. If the real networks are not connected originally, the giant component will be used instead.

- Undirected. Edges are not allowed to have direction. If the graph is directed, it will be converted into an undirected graph.

- All edges are unweighted. All the edges have no weights to be assigned.

- No multi edges are allowed. There will be at most one edge between two nodes.

- No loops are allowed. Nodes are not allowed to connect to themselves.

Definition 1.1 *(Graph/network) A graph or network G can be represented by three sets. A node set $V : \{V_1, V_2, V_3...V_n\}$ and an edge set $E : \{E_1, E_2, E_3...E_m\}$ and a function set ι such that $E \rightarrow V_\alpha \times V_\beta$ implies there is a connection between node V_α and node V_β.*

In non-mathematical words, a network or a graph is a collection of nodes and edges, each node can connect to other nodes via edges. The border between the term network and graph is very ambiguous, and network scientists use them interchangeably. There are many different types of graphs. A path graph contains n nodes and all nodes are connected in a sequence. A clique has all its nodes connect to all the other nodes. Since a clique is fully connected, this leads to a total number of $n(n-1)/2$ edges.

Definition 1.2 *(Subgraph) A subgraph G' is a slice of a graph G, such that $V' \epsilon V$, $E' \epsilon E$ and ι' matches two nodes belongs to V', i.e. $E' \rightarrow V'_\alpha \times V'_\beta$. All the edges must have their starting and ending points within the subgraph.*

Definition 1.3 *(Neighbours) If node i is connected to node j, node i is a neighbour of node j, vice versa. If node i is connected to node j, k, l, the node set $\{j, k, l\}$ is said to be the neighbourhood of node i.*

Definition 1.4 *(Distance) Distance $d_{i,j}$ in a undirected, unweighted graph is the minimum number of edges required to connect node i and j. Average distance $\langle d \rangle$ is defined as $\langle d \rangle = \frac{2}{n(n-1)} \sum_{i,j} d_{i,j}$.*

Referring to figure 3, distance between node 1 and 6 is 3, denoted by $d_{1,6} = d_{6,1} = 3$. Distance between node i and node j is equal to node j to node i: $d_{i,j} = d_{j,i}$. The average distance of the network is 1.53.
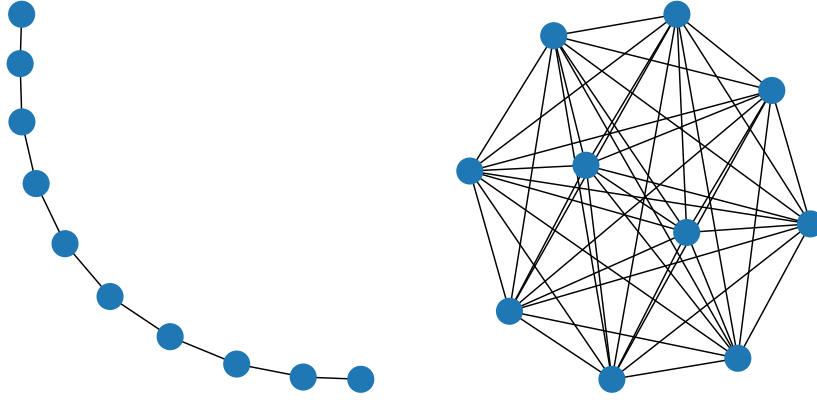
Figure 2: From left to right: A path with 10 nodes($n = 10$) and 9 edges($m = 9$); a clique with 10 nodes and 45 edges($m = 45$).
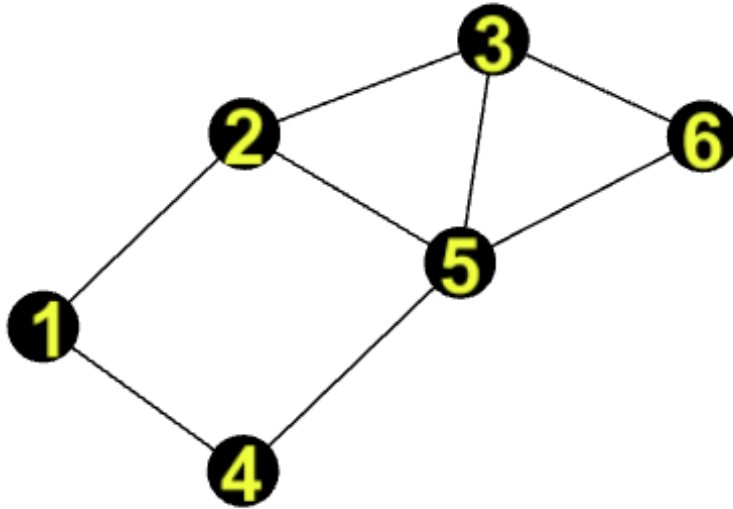
Figure 3: A simple network with 6 nodes and 8 edges.

A node i has degree k_i if it has k edges connected to other nodes. For figure 3, we have $k_1 = 2, k_2 = 3, k_3 = 3, k_4 = 2, k_5 = 4, k_6 = 2$. Thus, the total number of edges L in a network can be calculated:

$$L = \frac{1}{2} \sum_{i=0}^{i} k_i \tag{1}$$

Using equation 1, we can calculate the total number of links for figure 3 is equal to:

$$L = \frac{1}{2}(2 + 3 + 3 + 2 + 4 + 2) = 8 \tag{2}$$

An important parameter of a network is generated using degrees, which is average degree of a network:

$$\bar{k} = \frac{1}{N} \sum_{i=0}^{i} k_i \tag{3}$$

or, simply:

$$\bar{k} = \frac{2L}{N} \tag{4}$$

Applying formula 3 to figure 3, the average degree is:

$$\bar{k} = \frac{1}{6}(2 + 3 + 3 + 2 + 4 + 2) = \frac{8}{3} \tag{5}$$

or equivalently, using formula 4:

$$\bar{k} = \frac{1}{6}(8 * 2) = \frac{8}{3} \tag{6}$$

Definition 1.5 *(Clustering coefficient) For a given node i with degree k, the clustering coefficient is defined as:*
$C_i = \frac{2L_i}{k_i(k_i - 1)}$
where L_i represents the number of links exists between node i's neighbours.
Clustering coefficient C_i illustrates the proportion of edges exists between a node i's neighbours.

In figure 3, node 2 and node 4 are node 1's neighbours. There are no edges between node 2 and node 4, so $C_1 = 0$. On the other hand, node 3 has neighbours 2,5 and 6. There are 2 edges between node 2,5 and 6, $C_i = \frac{2*2}{3*2} = \frac{2}{3}$. Average clustering tends to suggest the average connectivity of the network.

Definition 1.6 *(Laplacian Matrix) A Laplacian matrix is defined as:*

$$L_{i,j} = \begin{cases} k_i & \textit{if i=j} \\ -1 & \textit{if an edge exists between node i and j, and i is not equal to j} \\ 0 & \textit{otherwise} \end{cases} \tag{7}$$

Diagonal elements $L_{i,i}$ suggests the degree of node i. A non-zero non-diagonal element $L_{i,j}$ implies a connected edge between node i and j. Laplacian matrix allows quick calculation of the graph properties and a visualisation of the network.

To evaluate different complexity measures, we need to introduce some graph models that are often used to model real world problems.

1.2 Random graphs

There are infinite numbers of real networks in the real world, but defining or observing all of them is not feasible. For simulations and comparisons, network scientists introduced the idea of random networks. They are also known as Erdős-Rényi network in honour of two mathematicians: Paul Erdős and Alfréd Rényi. They have important contributions to understand the properties of a random network[6].

There are two definitions of a random network:

- $G(n, p)$ network. A network with n nodes will be initialised, there will be at most $(n)(n-1)/2$ edges. Each edge will be instantiated with probability p. This approach brings a randomness property to the graph; number of edges m. The expectation of m is equal to $p(n)(n-1)/2$.

- $G(n, m)$ or $G(n, L)$ network. A network with n nodes will be initialised, m or L edges will be connected from a random node to another random node. Due to the non-randomness of $G(n, m)$ networks, they are used to simulate the behaviour of a random network in this thesis. They will be referred to a random network/random graph in this thesis unless stated otherwise.

Previously, the definition of clustering coefficient and average distance were introduced. For a given random graph, the clustering coefficient and average distance can be calculated using formulas.[7] The average clustering coefficient of a random graph is p, or $2m/((n)(n-1))$(number of instantiated edges divided by total number of possible edges). The clustering coefficient is used to illustrate the ratio between connected links and possible links between a node's neighbours. If there are k neighbours of a node, there can be at most $k(k-1)/2$ edges between the neighbours. In these $k(k-1)/2$ edges, only p of them will be instantiated. Thus, the ratio of connected edges and possible edges becomes $\frac{pk(k-1)/2}{k(k-1)/2} = p$. Additionally, average distance of a random graph $L_r \approx \frac{ln(n)}{ln(k)} \approx \frac{ln(n)}{ln(2m/n)}$. To be noticed, both parameters are expectations.

1.3   Rewiring

Network scientists desire more techniques to allow them to modify the properties and variables of a network. Network scientists would use a technique named rewiring to change the properties and parameters of a network, and monitor the change of parameters with respect to the rate of rewiring.[8] In this project, two simple rewiring techniques are used: single link rewiring and pairwise rewiring.

- Starting with an edge (u, v); with starting node u and ending node v. Single link rewiring will look for a node w that hasn't yet been connected to node u. Once w is found, edge (u, v) will be removed and a new edge (u, w) will be added to the network.

- Starting with two edges (u, v) and (x, y). Pairwise rewiring will remove both edges, and two new edges will be added: (u, y) and (x, v).

Single link rewiring tends to give higher randomness to the network, and pairwise rewiring preserves the degree distribution. Both rewiring techniques require a parameter p, which is the probability of rewiring for each edge. If $p = 1$, using single link rewiring will cause the network to become a random network. However, since pairwise rewiring preserves the degree distribution, $p = 1$ will not cause the network to become completely random.

1.4   Small-world

About 50 years ago, Standley Milgram[9] carried out a famous study in the interest of this question: how many intermediates are needed to pass a message between two irrelevant or distanced people? This is known as the small-world problem. As counter-intuitive as it may seem, the medium number of intermediates needed is only 5(an average of 6)[9]. This is not an undoubtedly experiment but it is almost impossible to determine the actual number of messengers needed in the real world. Nevertheless, this
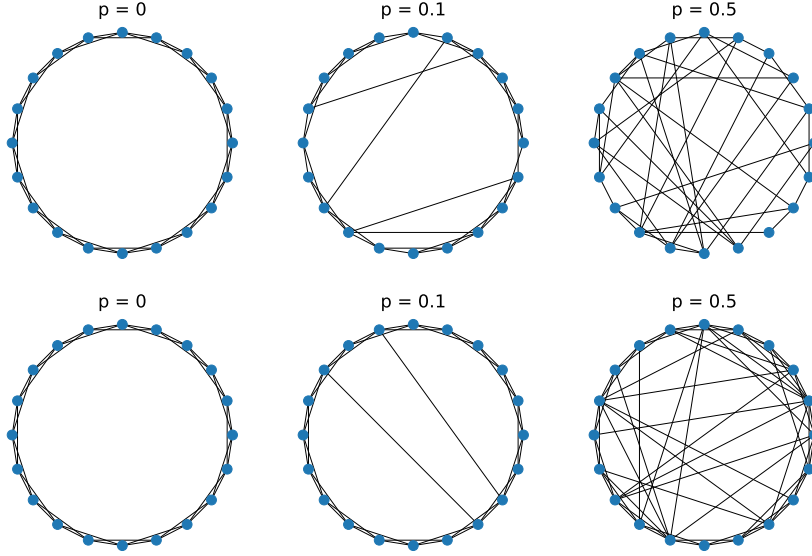
Figure 4: A demonstration of WS model(top) and NW model(bottom). The paramters are: $n = 20$, $k = 4$, $p = 0, 0.1, 0.5$.

number would be smaller than most peoples' expectations. Mathematically, the small world problem is the study of graphs with small average distance, because network scientists believe that in a real world network, the average distance is small with a high average clustering coefficient. Previously, we introduced the formula to calculate the average distance L_r of a random graph. Thus, if a graph has $L/L_r < 1$ and $C/C_r > 1$, this graph can be considered to have the small-world property.

A small-world network can be generated using a Watts-Strogatz(WS) model[10] or a Newman-Watts(NW) model(a variant of the WS model)[11]. Both models require three parameters: number of nodes n, number of connected closest neighbours k and rewiring probability p. The key of both models is rewiring. The graph starts with n nodes, each node is connected to $k(k-1$ if k is odd) nearest neighbours; $nk/2$ edges will be created. For each edge, there is a probability p that this edge will be rewired to another ending node(single link rewiring). While rewiring, the WS model removes the edge (u, v) and add a new edge (u, w). Thus, the number of edges stays the same. However, the NW model maintains the edge (u, v) and adding the new edge (u, w), causing the expectation of the number of edges after rewiring to be $nk/2 + pnk/2$. Rewiring will add short paths to the network, and cause the average distance to be exceptionally smaller. Suggested by Barabási[7], to obtain both high clustering and low average distance(properties of a small-world network), p should be between 0.001 and 0.1.

1.5 Scale-free network

A controversial topic of network science is whether real networks are usually scale-free[12]. To understand what is a scale-free network, we need to scope into the degree distribution of graphs.
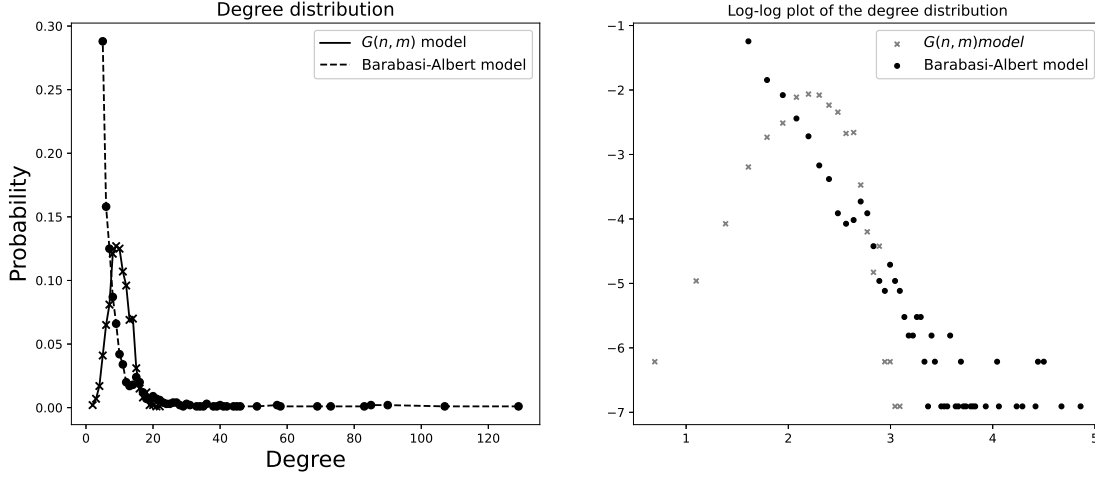
Figure 5: Degree distribution of a $G(n, m)$ graph with $n = 1000, m = 5000$ and a graph generated using Barabási-Albert model with $n = 1000, m = 5$.

Suggested by Barabási[7], the degree distribution of a random network is expected to follow a binomial distribution. However, it might not be the actual distribution of real networks. A controversial idea that hasn't yet been proven in the network science community is: are real networks' degree distribution follows a power-law distribution? A power-law distribution follows: $P(k) \sim k^{-\gamma}$, the parameter γ is typically in the range $2 < \gamma < 3$. If the degree distribution of a network follows a power-law distribution, the network is said to be a scale-free network. Even though there are counter-examples[13], many network scientists still believe that real networks are scale-free[14]. In order to further simulate the behaviour of real networks, Barabási introduced the Barabási-Albert(BA) model to create scale-free networks.[7].

The BA model requires two parameters: n and m. Initially, only one node is created and $n-1$ nodes will be added to the network consequently. Whenever a node is added to the network, it will connect m nodes. The logic of connection is the key to this model. Nodes are more likely to link to nodes with more edges than nodes with less edges. For instance, a node has been added to the network, this node will connect to a node with degree $k = 7$ with a higher probability than a node with degree $k = 3$. This logic of connection is called preferential attachment. Essentially, like in the real world, nodes are more likely to connect to nodes that have more impact on the network.[15] The BA model ensures most of the nodes have low degrees, whereas only a few nodes have exceptionally high degrees, as shown in figure 5. An ideal way to fit the power-law distribution is using the linear regression technique to fit the data in log-log scale.

1.6 Generating graphs

All graph models are utilized as below(unless specified):

- $G(n, m)$ random graphs/networks or random graphs/networks. With given n, m will be randomly selected between $n-1$ to $n(n-1)/2$ to create various networks with different number of m.

9

- For WS and NW graphs, three parameters are required. Given parameter n, k will be randomly selected between 1 and $(n-1)$ to give a larger range of m. p is also randomize, within the range 0.01 and 0.1 to simulate a small-world network as mentioned in section 1.4. By randomize two parameters, more diverse samples will be generated.

- BA graphs require n, m as parameters. They are randomised between 1 and $(n-1)$, so we can generate samples with a different number of edges.

2 Methods

2.1 Implemeted methods

In this project, 8 methods will be introduced, 7 of them are based on paper published[16][17], and a new measure MA_{RI} will be introduced. MA_{RI} is based on the idea of MA_g, and it also performs similarly.

To compare complexities, all complexities should be normalised within range $0 < complexity < 1$. Hence, finding the upper bound and lower bound of each complexity is essential, even though the lower bound for most measures is 0.

- Subgraph measures:

  - $C_{1e,st}$
  - $C_{1e,spec}$
  - $C_{2e,spec}$

- Product measures:

  - MA_g
  - MA_{RI}
  - Cr
  - Ce

- OdC (Entropy measure)

2.2 Different subgraph measures

Definition of a subgraph was introduced in section 1.1.1, so an intuitive idea of measure is to count the number of different subgraphs in a given graph. The question is: how to count subgraphs and how to determine whether they are identical or not? Different subgraphs can be created using edge deletion. There are two approaches in this project: removing one edge or remove two edges. Removing more edges is not optimal considering the complexity of calculations. Deleting one edge from a graph with m edges will result in m subgraphs, or $\binom{m}{2}$ using two edges deletion. After obtaining m or $\binom{m}{2}$ subgraphs, the next step is to find the number of unique subgraphs. Two identical subgraphs are said to be isomorphic. To determine isomorphism, there are two possible methods.

The first approach is to find out the number of spanning trees of a subgraph. A spanning tree contains all the nodes in a subgraph with the minimum number of edges. This can be calculated using Kirchkoff's

theorem[18]: number of spanning trees = cofactor of Laplacian matrix L. By counting different numbers of spanning trees, isomorphism can be determined: two subgraphs with the same m, n and the same number of spanning trees are isomorphic. This is a computationally cheaper approach with possible errors. A more complex but precise way to determine isomorphism is to compare the spectrum of the Laplacian matrix of subgraphs. Calculating and comparing both the Laplacian matrix and the signless Laplacian matrix(all entries are positive or 0) will offer a better result. If two subgraphs with the same spectrum on both the Laplacian matrix and signless Laplacian matrix, they are isomorphic. As mentioned, the complexity needs to be normalised to allow comparison. A good upper bound for both edge deletion method is $m_{cu} = n^{1.68} - 10$.[16].Three measures can be defined:

- $N_{1e,st}$, the number of different subgraphs with different number of spanning trees, after cutting one edge. The normalised complexity $C_{1e,st} = (N_{1e,st} - 1)/(m_{cu} - 1)$.

- $N_{1e,spec}$, the number of different subgraphs with different spectrums, after cutting one edge. The normalised complexity $C_{1e,spec} = (N_{1e,spec} - 1)/(m_{cu} - 1)$.

- $N_{2e,spec}$, the number of different subgraphs with different spectrums, after cutting two edges. The normailised complexity $C_{2e,spec} = (N_{2e,spec} - 1)/(\binom{m_{cu}}{2} - 1)$.

2.3 Potential problems and solutions of different subgraph measures

Problems were found during the implementation of different subgraphs. Possible solutions are also given for future studies.
Different subgraph measures are principally simple, but they are complex to compute, within at least $O(n^2)$ time[16]. This is not the only problem. An upper bound of the complexity $m_{cu} = n^{1.68} - 10$ was assumed by Kim and Wilhelm[16] to normalise the complexity. However, from the simulation, we found that this may not be the actual upper bound of the different subgraph measures.
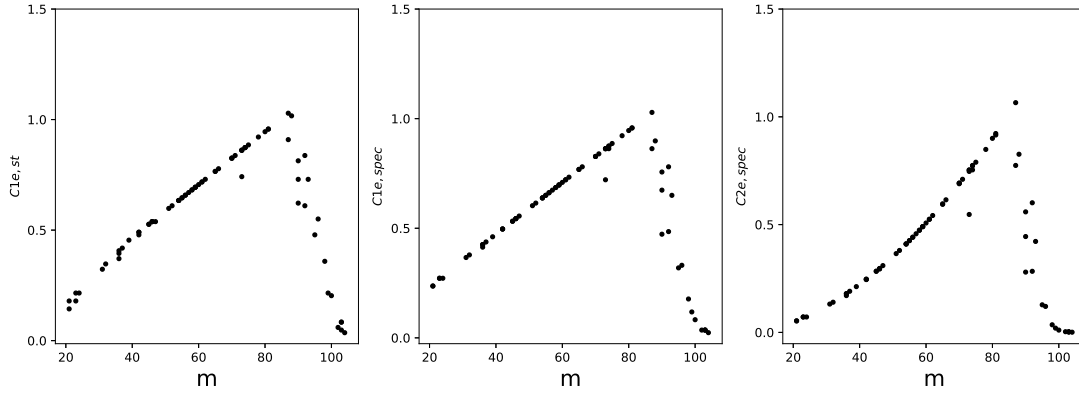
Figure 6: Different subgraph measure of 1000 $G(n,m)$ random graphs, with $n = 15$.

The complexity is abnormal for graphs with around 90 edges and 15 nodes as shown in figure 6. These exceptions could imply that the upper bound assumption m_{cu} is not correct, but there is another possible reason, which is the problem of the floating-point arithmetic.

On most machines today, numbers are represented in binary system[19]. For example, 0.2 is recorded as 0.00110011001100110011... in a binary system. This series is infinite, represented by $1 * 2^{-3} + 1 *$

11

$2^{-4} + 1 * 2^{-7} + 1 * 2 - 8....$ For obvious reasons, computer scientists don't want to work with infinite series. Therefore, the series is approximated. On a modern computer, the series is usually approximated to 63 digits, and an extra one digit represents the sign of the number. After approximation, the error could cause the equal operation to fail. A famous example is that for modern programming language or machine that operates this numbering system, 0.2+0.1 does not equal 0.15+0.15. As a result, the inaccurate comparison may cause mistakes and generates more different subgraphs than actual.

The core of different subgraph measure is to compare the cofactor ($C_{1e,st}$) or spectrum ($C_{1e,spec}$ and $C_{2e,spec}$) of a subgraph. Given the fact that the probability of a decimal number to appear in the spectrum is high, and the cofactor might exceed the upper bound of the numbering system[20]. The comparisons will be inaccurate with a high probability. There are three possible solutions:

- As suggested, errors will be made when approximated by the machine. An error threshold can be used when comparing spectrums and cofactors. For example, two numbers with relative error less than 1% can also be considered equal numbers. One disadvantage is the increase of complexity, taking more time and effort to compare the spectrums/number of spanning trees.

- Similarly, numbers can be rounded before comparison to avoid errors. This is used in the implementation of different subgraph measures, all cofactors and spectrums are rounded to the first ten significant figures. This solution requires less computation time than the first solution and reduce the number of misclassification. The drawback is that similar graphs can be considered as isomorphic graphs, this disadvantage also applies to the first provided solution, but with higher accuracy for large graphs. This solution may still cause complexities larger than 1, but it is the best solution considering the effort spent. Thus, this solution will be used in this project.

- Instead of using m_{cu} as the upper bound, m or $n(n-1)/2$ can be used for one-edge-deleted subgraph complexity and $\binom{m}{2}$ for two-edges-deleted subgraph complexity. This guarantees the normalisation and avoids the mistakes that are caused by the first two solutions. On the other hand, causing the complexity to be different.

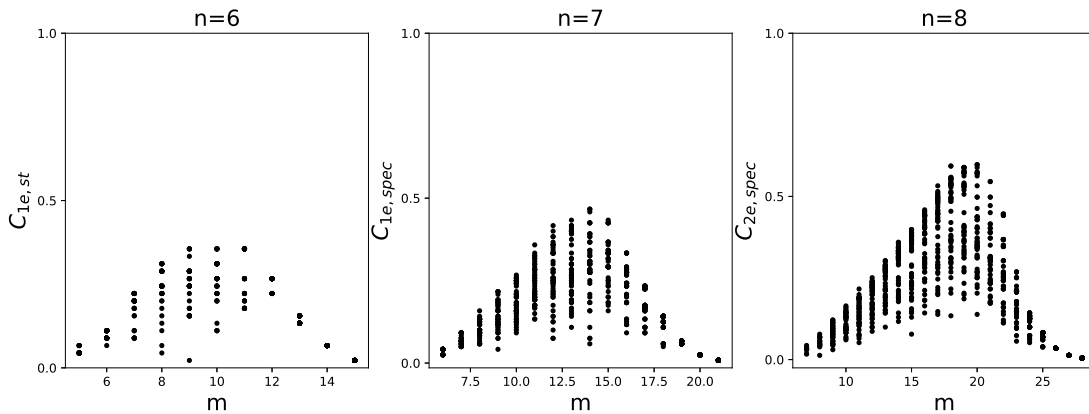A unique problem with $C_{2e,spec}$ is the value is not properly scaled for small graphs.

Figure 7: $C_{2e,spec}$ complexities of $G(n,m)$ random graphs for n = 6,7,8 respectively. For each m, 50 samples are generated

The upper bound of $C_{2e,spec}$ is 0.5 while $n \leq 7$. To have an upper bound at 1, the complexity values have to be scaled by 2. However, scales by two will cause the complexity to exceed 1 for larger graphs.

12

Thus, we stick to the original normalisation and scaling: $C_{2e,spec}$ will have an upper bound at 0.5 for $n = 7$, and less for smaller ns.

2.4   Product measure

The core idea of product measure is to assign low complexity at both extremes(path and cliques) and higher complexity to graphs with a medium number of edges.

2.4.1   Medium articulation for graphs(MA_g)

This measure is based on redundancy and mutual information of a graph.[21] The complexity before normalisation can be defined as the product of redundancy and mutual information $C = R * I$, with redundancy $R = \frac{1}{m} \sum_{i,j>i} ln(d_i d_j)$ and mutual information $I = \frac{1}{m} \sum_{i,j>i} ln(\frac{2m}{d_i d_j})$. The core of this measure is the variable $d_i d_j$. It is product of degree of two nodes. This variable will be maximised when the graph is fully connected(clique) and minimized when the graph is least sparse and highly ordered(path).

- Highest redundancy: $R_{clique} = 2ln(n-1)$

- Lowest redundancy: $R_{path} = 2(\frac{n-2}{n-1})ln(2)$

- Highest mutual information: $I_{path} = ln(n-1) - (\frac{n-3}{n-1})ln2$

- Lowest mutual information: $I_{clique} = ln(\frac{n}{n-1})$

Since the maximum and the minimum of redundancy and mutual information can be found, the normalisation technique can be applied to ensure the complexity MA_g is bounded by 0 and 1.

$$MA_g = MA_R * MA_I, \tag{8}$$

with:

$$MA_R = 4(\frac{R - R_{path}}{R_{clique} - R_{path}})(1 - \frac{R - R_{path}}{R_{clique} - R_{path}}) \tag{9}$$

$$MA_I = 4(\frac{I - I_{clique}}{I_{path} - I_{clique}})(1 - \frac{I - I_{clique}}{I_{path} - I_{clique}}) \tag{10}$$

2.4.2   Efficiency complexity(Ce)

Efficiency complexity is based on the efficiency of adding shorter path in a network.[22] As suggested in section 1.4, adding short paths will significantly shorter the average shortest distance in a network. However, adding more paths will cost energy. The ratio between shorten distances and cost of energy is defined as the efficiency. A complex graph should maintain short average distance and low energy used as possible. The efficiency is calculated as $E = \frac{1}{n(n-1)/2} \sum_{i,j} \frac{1}{d_{i,j}}$. To normalise the complexity, identify the lowest graph with lowest efficiency, a path, is: $E_{path} = 2/(n(n-1)) \sum_{i=1}^{n-1}(n-i)/i$. Consequently, the normalised complexity is $C_e = 4(\frac{E-E_{path}}{1-E_{path}})(1 - \frac{E-E_{path}}{1-E_{path}})$

2.4.3 Graph index complexity(Cr)

In section 1.1.1, Laplacian matrix was introduced. Cr is depending on another graph matrix which is called adjacency matrix. The adjacency matrix is essentially Laplacian matrix with all diagonal elements being 0. This records all the edges within the network. Using previous studies, all eigenvalues of the adjacency matrix are real.[23] The largest eigenvalue is also knows as the index r. To normalise the complexity, r is found out that $2cos(\pi/(n+1) \leq r \leq n-1)$.[23] Therefore, a normalised complexity can be constructed, which is given by: $Cr = 4c_r(1-c_r)$ with $c_r = \frac{r-2cos(\pi/(n+1))}{n-1-2cos(\pi/(n+1))}$.

2.5 Offdiagonal complexity(OdC)

A node-node link correlation matrix need to be constructed to calculate OdC.[17] The matrix element $c_{i,j}$ records the number of neighbours with degree $j \geq i$ of all nodes with degree i. OdC assigns higher complexity to graphs that their nodes have no preferences on their neighbours degree. A vector a_n is calculated by summing up all the rows in the matrix. The normalised OdC is calculated as:
$OdC = -(\sum_{n=0}^{k_{max}-1} A_n ln(A_n))/ln(n-1)$ with $A = a_n/\sum_{n=0}^{k_{max}-1} a_n$(probability of each a_n).
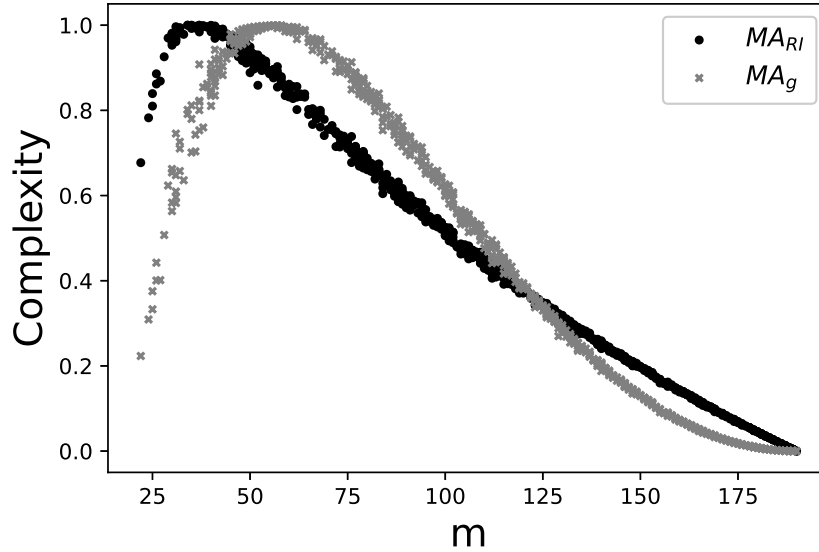
2.6 MA_{RI}

Figure 8: MA_g and MA_{RI} complexity of $G(n,m)$ networks, all networks have 20 nodes and 1000 samples with random m have been generated.

MAg measure is a product measure, which assigns higher complexity to graphs with medium number of edges and lower complexity at both extremes (path and clique). Using the product of redundancy R

14

and mutual information I, with normalisation, MAg is defined as[16]:

$$R = \frac{1}{m} \sum_{i,j>i} ln(d_i d_j)$$

$$I = \frac{1}{m} \sum_{i,j>i} ln(\frac{2m}{d_i d_j}) \quad (11)$$

$$MA_R = 4(\frac{R - R_{path}}{R_{clique} - R_{path}})(1 - \frac{R - R_{path}}{R_{clique} - R_{path}})$$

$$MA_I = 4(\frac{I - I_{clique}}{I_{path} - I_{clique}})(1 - \frac{I - I_{clique}}{I_{path} - I_{clique}}) \quad (12)$$

$$MA_g = MA_R * MA_I$$

I can be written as:

$$I = \frac{1}{m} \sum_{i,j>i} ln(\frac{2m}{d_i d_j})$$

$$I = \frac{1}{m}(\sum_{i,j>i} ln(2m) - \sum_{i,j>i} ln(d_i d_j)) \quad (13)$$

$$I = \frac{1}{m} \sum_{i,j>i} ln(2m) - \frac{1}{m} \sum_{i,j>i} ln(d_i d_j)$$

$$I = ln(2m) - R \quad (14)$$

$R_{path}, R_{clique}, I_{path}$ and I_{clique} represent the lowest redundancy, highest redundancy, highest mutual information and lowest mutual information of graphs with fixed n respectively. The equations can be found in section 2.4.1. Kim and Wilhelm suggested that network scientists may use $C = (R - R_{path})(I - I_{clique})$ as a complexity measure, however, the upper bound cannot be found to normalise the complexity [16]. From our study, an upper bound of C can be calculated analytically.

Assuming the upper bound C_{max} can be found, $0 < C/C_{max} < 1$. As suggested in equation 14, $I = ln(2m) - R$, we can rewrite the complexity equation:

$$C = (R - R_{path})(ln(2m) - R - I_{clique}) \quad (15)$$

$$C = -R^2 + (ln(2m) - I_{clique} + R_{path})R + (-R_{path}ln(2m) + R_{path}I_{clique}) \quad (16)$$

As shown in equation 16, the complexity function is a quadratic function with only variable R, which denotes, there is one and only one extreme. As a product measure, the extreme should be a maximum. To find the maximum, differentiates the function with respect to $R(R_{max})$ where the function's slope is 0:

$$\frac{dC}{dR} = -2R_{max} + ln(2m) - I_{clique} + R_{path} = 0 \quad (17)$$

$$R_{max} = \frac{ln(2m) - I_{clique} + R_{path}}{2} \quad (18)$$

Even without assumption, $d^2C/dR^2 = -2$ implies the extreme is a maxima. As C_{max} is found, substitutes equation 18 into equation 15:

$$C_{max} = (R_{max} - R_{path})(ln(2m) - R_{max} - I_{clique})$$

$$C_{max} = (\frac{ln(2m) - I_{clique} + R_{path}}{2} - R_{path})(ln(2m) - \frac{ln(2m) - I_{clique} + R_{path}}{2} - I_{clique}) \qquad (19)$$

$$C_{max} = (\frac{ln(2m) - I_{clique} - R_{path}}{2})(\frac{ln(2m) - R_{path} - I_{clique}}{2})$$

$$C_{max} = \frac{(ln(2m) - I_{clique} - R_{path})^2}{4} \qquad (20)$$

Thus, using equation 20, a new normalized measure MA_{RI} can be defined using C/C_{max}:

$$MA_{RI} = \frac{4(R - R_{path})(I - I_{clique})}{(ln(2m) - I_{clique} - R_{path})^2} \qquad (21)$$

As shown in figure 8, MA_{RI} gives higher complexity to sparser graphs but less complexity when approaching a medium number of links. Additionally, it decreases more linear and smoothly compare to MA_g.

3 Result

3.1 Complexity measures on small random graphs

To validate all the implemented measures, all measures are applied on $G(n, m)$ random graphs, with $n = 7$, and 50 samples are generated for each m. As shown in figure 9, we reproduced results as Kim and Wilhelm did in [16]. Except $C_{2e,spec}$, as mentioned in section 2.3, there is a scaling problem. Most of the methods reach their maximum briefly before the medium number of edges. Low complexities are given to highly connected graphs and sparse graphs as expected.

Different subgraph measures perform similarly when the graph is neither sparse nor strongly connected, there is a big difference between the maximum and minimum with the same m. Thus, it is difficult to predict the complexity of a graph with given m and n. The highest complexity is reached at $m = 15$ for $C_{1e,st}$ and $C_{1e,spec}$ and $m = 14$ for $C_{2e,spec}$. There is a miss in the plot: $C_{1e,spec}$ and $C_{2e,spec}$ plot does not contain a data point at (6,0). There is a very small probability for $G(n, m)$ model to generate a star graph(n-1 nodes are connected to 1 node, in total of n-1 edges), which will result in 0 complexity using $C_{1e,spec}$ and $C_{2e,spec}$ measure. Uses different subgraph measures for small graphs is optimal as it is relatively independent of m, but not for large graphs due to its complexity.

OdC is based on the node-node link correlation matrix of a graph.[17] Thus, it spreads across the space and has little relationship with m. OdC assigns a lot of graphs with six edges high complexity than desired. OdC is "hierarchy sensitive", it may not creates a big difference between graphs when the graphs are considerably small.

All 4 product measures are similar, gives higher complexity value at a medium number of edges and less at both extremes. There is a very small difference between graphs with the same number of edges. Ce and Cr tend to give the highest complexity to graphs with exactly $n(n - 1)/4$ edges, and MA_{RI} and MA_g reach their maximum before a medium number of edges as expected. Product measures are highly dependent on m, such that the complexity of a graph can be approximated solely based on m and n. Network scientists may use machine learning techniques to estimate the complexity of a graph using m and n in a small amount of time. On the other hand, product measure may not be optimal because a complexity measure should not solely base on m and n, but the overall structure of a network.
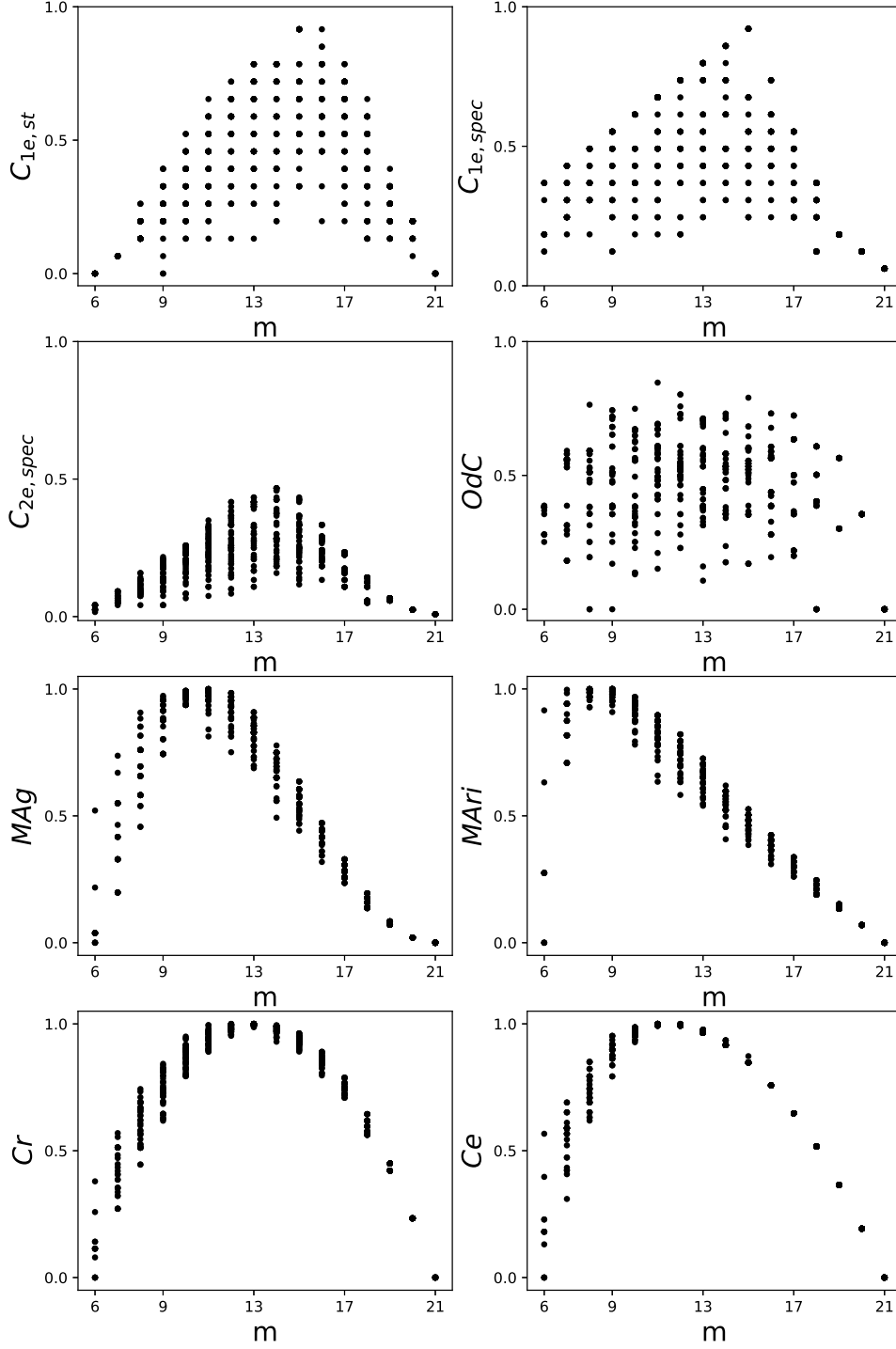
Figure 9: Complexity of graphs generated using $G(7, m)$ model, 50 samples are generated for each m. Methods from top-left to bottom-right are: $C_{1e,st}$, $C_{1e,spec}$, $C_{2e,spec}$, OdC, MAg, Cr, Cr and $MAri$.
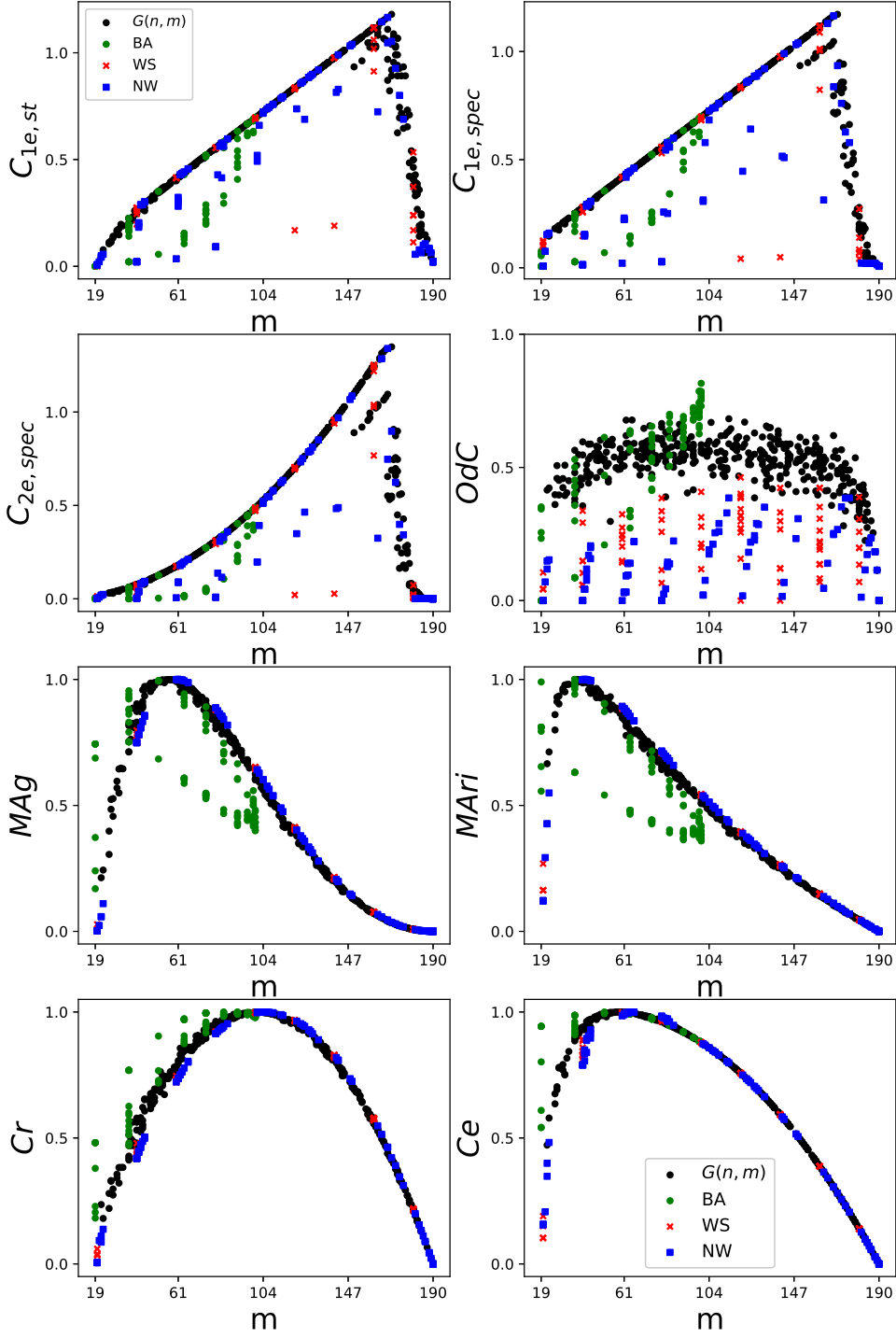
Figure 10: Complexity of 500 $G(n, m)$ graphs, 100 BA graphs, 100 WS graphs and 100 NW graphs, with $n = 20$. Graphs are generated according to section 1.6.

3.2 BA,WS and NW model

As informed in section 2.3, different subgraph measures have normalisation problems and the complexities would exceed one.

Surprisingly, different subgraph measures and product measures are struggling to separate random graphs, WS graphs, and NW graphs. Only OdC separates random graphs and WS, NW model by giving random graphs higher complexity than WS and MW model with fixed m. This is because OdC awards graphs with complicated degree correlations.

More fascinating results are obtained from BA graphs. Different subgraph measures assign lower complexity to BA graphs compare to random graphs. This can be caused by preferential attachment. Preferential attachment ensures most nodes have low degrees and builds hubs(nodes with high degrees) in the graph. After cutting an edge/two edges between hubs and nodes with small degrees, there is a high chance an isomorphic subgraph can be found, thus lower the complexity of the graph. Specifically, subgraphs resulted by cutting the edge between hubs and nodes with small degrees are very similar and occasionally isomorphic.

MA_g and MA_{RI} perform similarly by assigning BA graphs lower values towards a medium number of links. Both measures are depending on the variable $\sum_{i,j>i} d_i d_j$. BA graphs usually have less $\sum_{i,j>i} d_i d_j$ because they are highly structure, causing less sum than a random graph. Contrarily, Ce and Cr cannot distinguish BA graphs and random graphs. Ce is based on the efficiency of a graph, a highly complex graph should have a small average distance with not too many edges simultaneously. BA graph does not perform differently than random graphs in Ce measure.

3.2.1 Configuration model

Figure 11: Complexities of graphs generated using configuration model, $n = 50$. Results are average of 50 simulations.

As introduced in section 1.4, a network is said to be scale-free if its degree distribution follows a power-law distribution with $2 < \gamma < 3$. To observe how the change of γ would affect the complexity

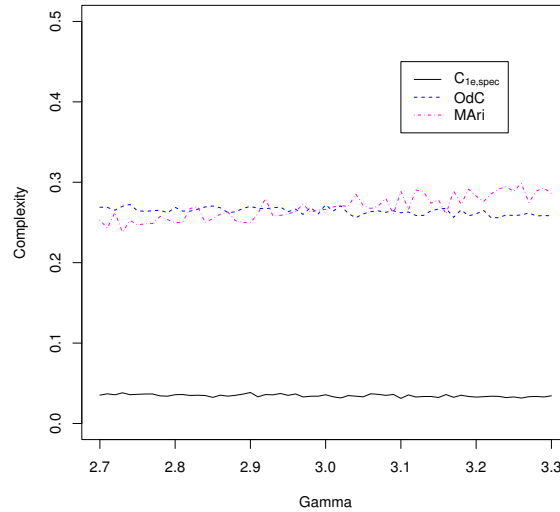of a network, we need a model that can generate the graph with given γ. A configuration model [24] is able to turn a given degree distribution into a graph, which has the exact degree distribution as the given degree series. OdC and $C_{1e,spec}$ didn't change much by varying γ. MA_{RI} increases briefly as it is very sensitive to the change of degrees of nodes. Overall, varying γ does not impact the complexity by a lot. Due to the limitation of our implementation, we can only generate a degree distribution within the range $2.7 < \gamma < 3.3$, by varying n and γ in a larger range and adding a new variable n in the model, it can be more inspirational.

3.3   Complexity correlation

Different types of measures focus on different properties/parameters of a network, monitor the correlation between measures to enable further understanding of the properties/parameters each measure is highlighting on. Additionally, network scientists may use more than one complexity measure on a network to determine whether they are truly "complex" or not. Combining complexity measures on networks will allow network scientists to comment on the network complexity from a different aspect. For these reasons, we choose three measures($C_{1e,st}$,OdC and MA_{RI}) and monitored their behaviours on random and BA graphs. The reason BA graphs are added is to investigate whether distinguish BA graphs and random graphs is possible or not.
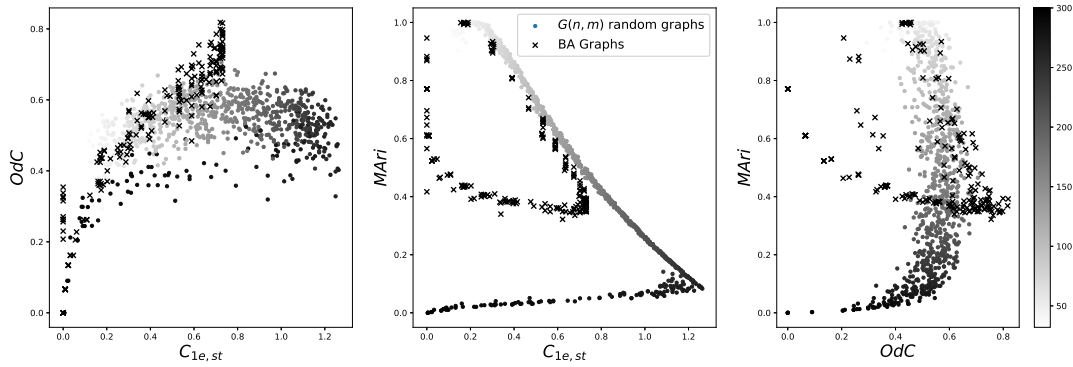
Figure 12: Correlation of complexity measures on 1000 random graphs and 200 BA graphs with $n = 25$. Generated according to the rules stated in section 1.6. Colorbar represents change of m: higher the m, darker the data points.

As figure 12 suggests, the correlation between complexity measures are difficult to observe, since they are trying to address different things. As $C_{1e,st}$ value increases, OdC also increases. We indicated in section 3.1, m is not an important factor for OdC. On the other hand, $C_{1e,st}$ is highly based on m, the complexity increases linearly with m, and then complexity drops quickly after reaching the maximum. By constructing a correlation between $C_{1e,st}$ and OdC, separate BA graphs and random graphs are not possible.

Since both complexity measures are heavily dependent on m, thus we can observe a linearly decreasing trend. We can detect a unique distribution of the BA graphs' complexity: a shape between ellipse and half-moon. This is because both MA_{RI} and $C_{1e,st}$ assign lower complexity values to some of the BA graphs, comparing to other BA graphs with the same number of edges. Therefore, distinguish BA graphs and random graphs is difficult.

The correlation between OdC and MA_{RI} of the random graph seems simple: as m decrease, MA_{RI} decrease speedily but OdC change unnoticeably until the graph is highly connected. On the other hand, the correlation between OdC and MA_{RI} on BA graphs perform slightly differently. As suggested, MA_{RI} distributes lower complexity to some BA graphs, and OdC is very consistent. In section 3.2.1, we did not observe a big change of complexity by varying γ. To observe a better result, using a larger graph is optimal($n = 1000$), but due to the computation cost, they are not used here.

3.4 Complement graphs

The definition of a complement graph is fairly simple: the complement graph contains all the edges that are not in the original graph. To ensure complexity measures can be successfully applied, graphs that will cause the complement graphs to be disconnected will be removed. Thus, in theory, the original graphs and the complement graphs are not extreme graphs. Analysing the complexity correlation between the original graph and the complement graph gives us more inspiration of the measures. We have choose three different measures with different types: OdC, MA_{RI} and $C_{1e,spec}$. OdC seems to be very symmetric
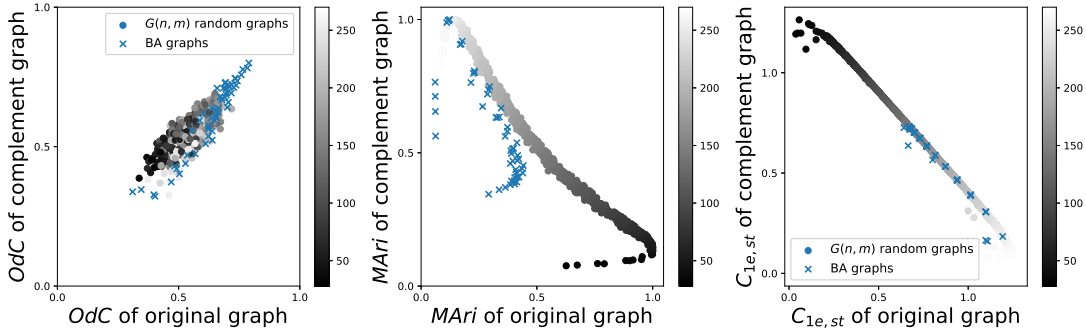
Figure 13: Complexities of the original graphs and complement graphs with $n = 20$. There are two types of graphs are generated: 500 $G(n, m)$ random graphs and 100 BA graphs.

with respect to m. Since taking the complement graph is reversing the degree distribution, the consistency of node-node link relation is preserved. Causing the complexity value for the original graph equal to the complement graph. As mentioned MA_{RI} is highly based on the number of edges, so observing a linear function is not a surprise. This also applies to BA graphs. Similar to MA_{RI}, $C1_{1e,st}$ is keener to m, causing another negative correlation between the original graph and the complement graph.

3.5 Applying MA_{RI} on real networks

To test the new measure MA_{RI}, six real networks are collected. To ensure comprehensive evaluation, various types of networks are used. Additionally, bus networks in 6 cities are collected and converted to networks from raw. Bus networks are different to other real networks. Bus networks contain an extraordinary amount of nodes with degree 2. For simplification and general interest, bus networks are modified. In modified bus networks, all nodes with $k = 2$ are removed, whereas the edges are preserved. For instance, node b is only connected to a and c. Node b will be removed from the network and a new edge (a, c) will be added to the network. The modification will be iterated multiple times until

bus networks are free from nodes with degree $k = 2$. The modification will significantly decrease the distance of bus networks. Moreover, generated graphs are also added to be evaluated and compared.

To be mentioned, for convenience, real networks are referred to non-bus networks.

Label	Name	Type	n	m	L	L_r	MA_{RI}	OdC
		Real networks						
1	Dolphins[25]	Animal interaction	62	159	3.357	2.524	0.999	0.517
2	PDZBase [26]	Protein interaction	161	209	5.326	5.326	0.824	0.310
3	Hamsterster[27]	Online social network	874	4003	3.217	3.058	0.963	0.532
4	Roget's Thesaurus [28]	Synonym network	994	3640	4.075	3.466	0.960	0.392
5	Flight[29]	Flight network	3397	19230	4.103	3.350	0.948	0.525
6	UK train [30]	UK train network	2490	4377	10.384	6.220	0.664	0.233
		Bus networks						
7	London[30]		8653	12285	32.338	8.687	0.38	0.127
8	Paris[31]		10644	12309	47.631	11.059	0.173	0.065
9	Berlin[31]		4316	5869	33.284	8.366	0.358	0.134
10	Sydney[31]		22659	26720	36.131	11.688	0.173	0.064
11	Detroit[31]		5683	5946	70.513	11.708	0.062	0.020
12	Beijing[32]		9249	14058	27.891	8.214	0.441	0.167
		Modified Bus						
13	London		3417	6018	18.308	6.462	0.553	0.128
14	Paris		2762	4301	15.386	6.975	0.468	0.106
15	Berlin		1662	2941	18.36	5.867	0.586	0.149
16	Sydney		4834	8358	17.665	6.838	0.49	0.089
17	Detroit		295	483	6.341	4.794	0.643	0.117
18	Beijing		4072	8325	14.864	5.902	0.64	0.195
		Generated networks						
19	G(n,m) random network($n = 875, m = 4000$)		875	4000	3.313	3.061	0.970	0.330
20	WS network($n = 875, k = 10, p = 0.05$)		875	4375	5.000	2.942	0.974	0.123
21	NW network($n = 875, k = 10, p = 0.05$)		875	4587	5.075	2.883	0.980	0.089
22	BA random network($n = 875, m = 5$)		875	4350	2.922	2.950	0.999	0.358

Table 1: Label,description and parameters of used networks

After careful consideration, average distance ratio L/L_r is chosen to be the variable showing on x-axis in figure 14. Average distance is a more important parameter to consider because it can identify how "small-world" the network is. As suggested, all the bus networks have a significantly higher average distance ratio than real networks. Even after being modified, the average distance ratios are still relatively high. But higher average distance ratio brings less MA_{RI} complexity. We can suggest a negative correlation between the average distance ratio and the MA_{RI} complexity. There is an exception of real networks, which is the UK train network. Considering the similarity between bus networks and train networks, it is not surprising. Generated graphs also behave similarly to standard real networks; a low average distance ratio with high complexity, as they are intended to simulate the behaviour of standard real networks. In theory, "Flight" is also a public transport system, so it should behave similarly to bus networks or the train network. However, the "Flight" network performs, unlike transport networks. Flight networks are
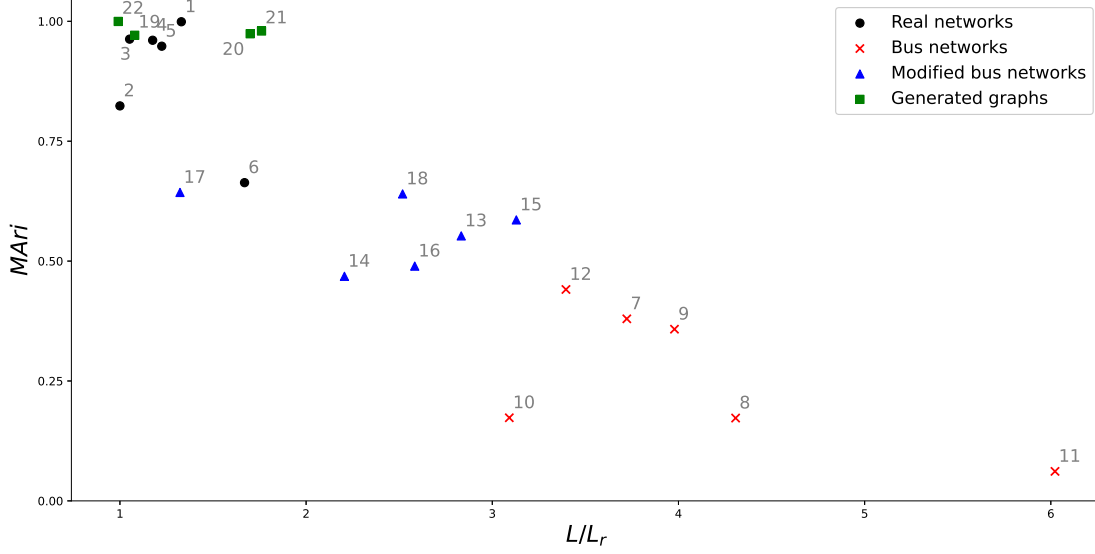
Figure 14: MA_{RI} complexity of real networks, bus networks, modified bus networks and graphs generated by graph models, labelling and description can be found in table 1.

designed to be robust[33] to be functional under severe weather/accident. Unlike trains or buses, flights are less limited by physical paths and routes.

To discuss the cause of the increase of MA_{RI} complexity after modification, we need to consider the only parameter MA_{RI} focusing on(with fixed m and n). $\sum_{i,j>i} d_i d_j$ is the only variable that affects the complexity of a graph. The essence of the measure is calculating the average $d_i d_j$. By removing nodes with degree 2, the average $d_i d_j$ will be increased and leads to an increase of the MA_{RI} complexity.

3.6 Rewiring on real networks

As recommended in section 1.3, rewiring is an important technique to monitor the behaviour of a network. Hence, we rewired real networks and modified bus networks to record their behaviour. Both single link rewiring and pairwise rewiring will be used. The reason we are going to use OdC instead of MA_{RI} is that pairwise rewiring will keep the degree relationship, and MA_{RI} will stay unchanged.

How we rewire graphs:

- The rewiring is done gradually. Initially, graph G will be rewired with probability $p = 0.05$. After recording the results, we rewire it with probability $p = 0.05$ again. This step will be repeated 20 times.

- Both rewiring have been simulated on all graphs more than ten times("dolphins" and "PDZBase" have been rewired more than 100 times, as they are relatively small) to generate more consistent results.

From figure 15, we can conclude that most real networks behave similarly using rewiring. As mentioned in section 1.3, single link rewiring results in higher randomness, because it destroys the degree distribution. In theory, when $p = 1$, the graph becomes a random network. OdC decreases significantly

23

Figure 15: Change of OdC complexities respect to change of rewiring probability with an error bar(one standard deviation).

with the increase of rewiring probability for real networks, whereas the change of complexity is not large for pairwise rewiring. This is b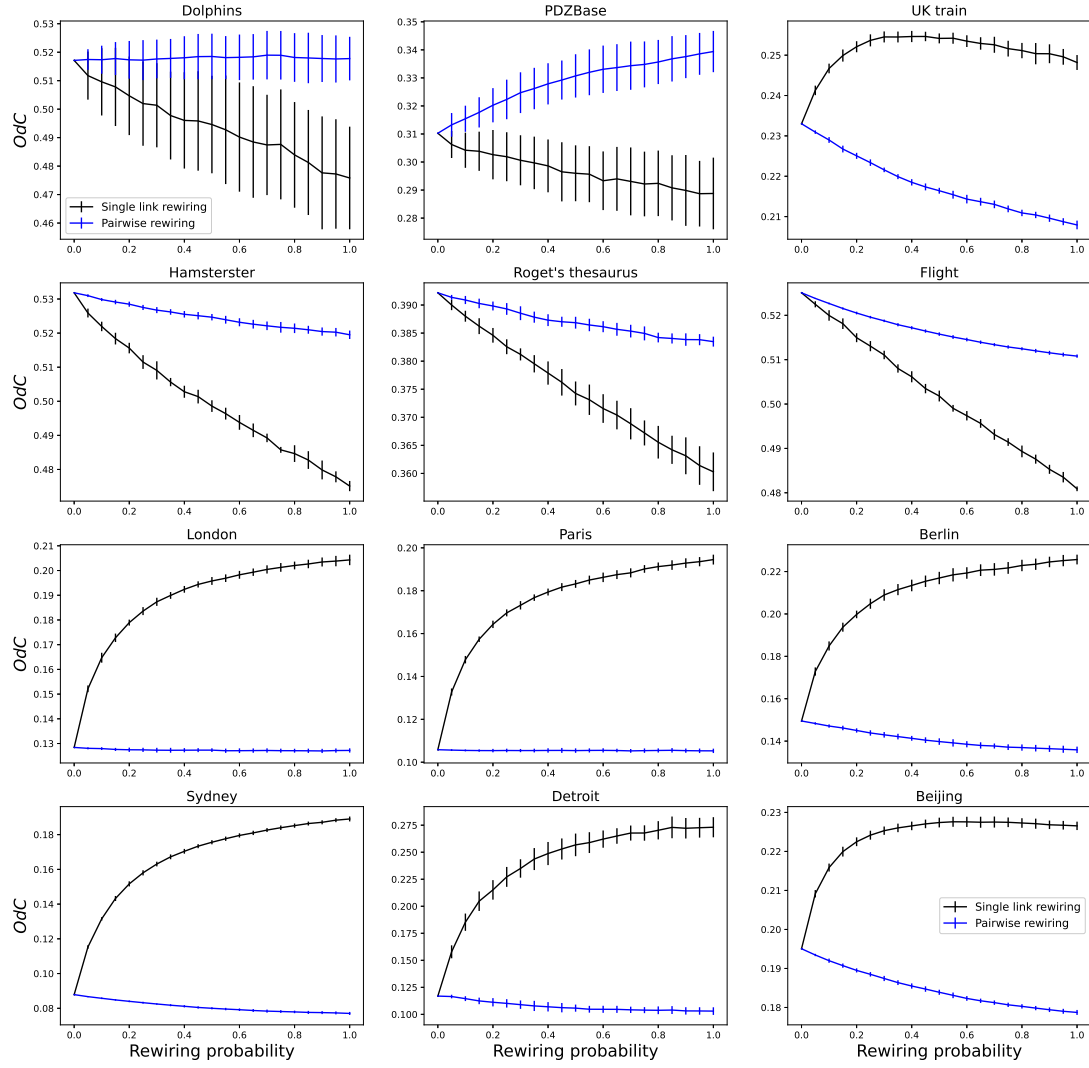ecause OdC is highly sensitive to degree relationship, but pairwise rewiring does not change the degree relationship. Also, the error bar for "dolphins" and "PDZBase" is large. This is because they are small networks, rewiring will make a larger impact than larger graphs.

There is an exception in real networks; the UK train network performs similar to bus networks rather than real networks. As public transportation networks, single link rewiring will cause the complexity to increase. Generally, the increase of complexity becomes small after $p = 0.4$; almost half of the links have been rewired. As shown in table 1, OdC complexities are very low for all the bus networks. As introduced by Claussen[17], OdC assigns high complexity to graphs that have no preference for the degree of their neighbours. After destroying the degree correlation using single link rewiring, the OdC complexity will increase.

In contrast, pairwise rewiring will cause the complexity to decrease briefly like real networks, with similar reasons.

4 Conclusion and further study

In this thesis, we investigated and compared different measures. The different complexity measures focus on different properties and parameters of a graph. For example, different subgraph measures focus on the general structure of subgraphs, lead to high complexity, and they are not feasible to apply different subgraph measures on large networks. In addition, the floating-point arithmetic is imperfect when dealing with large numbers and decimals. On the other hand, product measures are fairly simple in terms of calculation time. For example, Cr is the easiest complexity measure, which can be calculated in $O(n)$ time[16]. A drawback of product measures is that they are highly depending on m, creating smaller differences than other complexity measures with fix m and n. OdC measure can distinguish random graphs and graphs generated using BA,WS and NW model, and within a relatively small amount of time. However, the border of complexity value is wide. We suggest using the most suitable measure, depending on which specific parameters or properties are most important in the problem.

Additionally, we constructed a new measure MA_{RI} based on the idea of MA_g. MA_{RI} assigns higher complexity to sparser graph than MA_g. To be noticed, to calculate MA_{RI}, R_{clique} and I_{path} are not required, but m is involved in the calcualtion.

We also compare the difference between real networks and bus networks, and we investigated the unique property of bus networks: high average distance with low complexity, which is the opposite of a small-world network. This causes average lower complexities than other real networks.

There are more complexity measures [34][35] for further studies and researches. We encourage studies and applications of the measures on the different types of graphs and monitor the behaviour of different complexity measures. Further studies on complexity measures could help the network science community to build a comprehensive, robust and applicable complexity measure that everyone can agree on.

An unexpected fact was observed: the degree-based measures(OdC,MA_g,MA_{RI}) generally assign relatively high complexity to very sparse graphs, except Ce. We are not sure whether this is a coincidence or not, but this can be further study.

Overall, we hope our result can stimulate more studies on network complexity measures. The definition of complexity is already difficult to be determined. A good complexity measure should be able to distinguish: random networks, small-world networks, scale-free networks and real networks. In addition, the highest complexity to graphs with the number of edges is slightly less than the medium. Our

work provides useful aspects and observations to build an ideal complexity measure in the future.

A   Data

All real networks' citations can be found in the reference. All networks required pre-processing. Using Gephi is optimal. They need to be converted into the following form: a CSV file with two columns: "source" and "target", which store all the starting nodes and the ending nodes of all the edges. All processed real networks should be placed in "real_networks/processed" and processed networks in "bus/-processed". The version of programming languages and site packages are:

- Python 3.8.8

- NetworkX 2.6.3

- Matplotlib 3.4.3

- Numpy 1.21.2

- Pandas 1.3.3

- Scipy 1.7.1

- R 4.1.1

- plotly 4.9.4.1

- ggplot2 3.3.5

- igraph 1.2.6

- reticulate 1.20

If you prefer, you can find all relevant files that are used to build this project on my Github repository, including the Python virtual environment and R lock file which allow you to run the same version of packages: `https://github.com/yipeizhao/thesis_support_files`.

B   utilities.py

```python
import networkx as nx
import random
from random import randint
import pandas as pd
import numpy as np
import collections
from scipy.stats import pearsonr
from itertools import combinations_with_replacement
import math
```

```
from math import log
import matplotlib.pyplot as plt


#    Generates a list of graphs and their corresponding parameter.
     Notices the function will only return connected graphs,
     disconnected graphs will not be returned. Thus the actual number
     of graphs returned will be less than expected.
#    Parameters:
#    n: number of nodes
#    use_all_m: if this is true, then the function will generates
     samples using all the edge numbers * sample_number
#    sample_number: number of samples for each edge if use_all_m =
     True
#    Otherwise the sample_number defines how many graphs will be
     returned in total
def random_networks(n=7,use_all_m = True,sample_number = 50):
    #    Create a list to record all the graphs
    graph_list = []
    #    Max number of edges.
    m = (n*(n-1))/2
    m = int(m)
    #    Create a df to store relevant information
    column_names = ["Number_of_edges","Average_degree",
                    "Average_distance",
                    "Average_clustering"]
    #    Filling the df with zeros
    if use_all_m == True:
        zero_list = [float(0)]*(sample_number*(m+1))
        df = pd.DataFrame(columns = column_names)
        for item in column_names:
            df[item] = zero_list
    else:
        zero_list = [float(0)]*(sample_number)
        df = pd.DataFrame(columns = column_names)
        for item in column_names:
            df[item] = zero_list
    #    Creating graphs using different parameters
    #    count record the actual number of graphs generated by the
         function
    count = 0
    if use_all_m == True:
        for i in range(m+1):
            for j in range(sample_number):
                temp_graph = nx.gnm_random_graph(n,i)
                if nx.is_connected(temp_graph):
```

```
                    graph_list.append(temp_graph)
                    df["Number_of_edges"][count] = i
                    df["Average_degree"][count] =  (2*i)/n
                    df["Average_clustering"][count] = nx.
                        average_clustering(temp_graph)
                    df["Average_distance"][count] = nx.
                        average_shortest_path_length(temp_graph)
                    count  +=1


    else:
        for i in range(sample_number):
            edge_number = randint(0,m)
            temp_graph=nx.gnm_random_graph(n, edge_number)
            if nx.is_connected(temp_graph):
                graph_list.append(temp_graph)
                df["Number_of_edges"][count] = edge_number
                df["Average_degree"][count] =  (2*edge_number)/n
                df["Average_clustering"][count] = nx.
                    average_clustering(temp_graph)
                df["Average_distance"][count] = nx.
                    average_shortest_path_length(temp_graph)
                count +=1
    return graph_list, df
# Return a list of subgraphs of the graph G by taking an edge off
    the graph
def subgraph_one_edge_deletion(G):
    subgraphs = []
    for edge in list(G.edges):
        temp_graph = G.copy()
        temp_graph.remove_edge(edge[0], edge[1])
        subgraphs.append(temp_graph)
    return subgraphs
# Return the number of spanning trees of a graph.
# This calculation is based on the Kirchhoffs theorem, which is:
    number of ST = det(reduced Laplacian matrix)
# A reduced Laplacian matrix is the Laplacian matrix with a random
    row i and column i to be removed
def number_of_ST(G):
    L = nx.laplacian_matrix(G)
    L = L.todense()
    remove_i = randint(0,L.shape[0]-1)
    L = np.delete(L,remove_i,0)
    L = np.delete(L,remove_i,1)
    sign, det = np.linalg.slogdet(L)
    det = sign * math.exp(det)
    det = int(det)
```

```
        return det
#    Return number of isomorphic subgraphs of a graph
#    After taking all the one-edge-deletion subgraphs, investigates
   the number of different ST will remove isomorphic subgraphs
def isomorphic_graphs(G):
    subgraphs = subgraph_one_edge_deletion(G)
    ST_result = []
    for graph in subgraphs:
        ST_result.append(number_of_ST(graph))
    unique_ST = []
    unique_subgraphs = []
    ST_result = [str(item)[:10] for item in ST_result]
    for i in range(len(subgraphs)):
        if ST_result[i] not in unique_ST:
            unique_subgraphs.append(subgraphs[i])
            unique_ST.append(ST_result[i])
    return unique_ST
#    Generates a list of BA random graphs
#    Parameters:
#    n = number of nodes
#    sample_number = Number of samples
def BA_random_graphs(n, sample_number):
    graphs = []
    for i in range(sample_number):
        m=randint(1,n-1)
        temp_graph = nx.barabasi_albert_graph(n,m)
        if nx.is_connected(temp_graph):
            graphs.append(temp_graph)
    return graphs
#    Generates a list of WS random graphs
#    Parameters:
#    n = number of nodes
#    sample_number = Number of samples
def WS_random_graphs(n, sample_number):
    graphs = []
    for i in range(sample_number):
        p= random.uniform(0.001,0.1)
        k= random.randint(2,n)
        graphs.append(nx.watts_strogatz_graph(n,k,p))
    return graphs
#    Generates a list of NS random graphs
#    Parameters:
#    n = number of nodes
#    sample_number = Number of samples
def NW_random_graphs(n, sample_number):
    graphs = []
```

```python
    for i in range(sample_number):
        p= random.uniform(0.001,0.1)
        k= random.randint(2,n)
        graphs.append(nx.newman_watts_strogatz_graph(n,k,p))
    return graphs
# Return a list of subgraphs by taking two edges off from the graph
def subgraph_two_edge_deletion(G):
    subgraphs = []
    remove_edges = np.linspace(0,len(G.edges)-1,len(G.edges))
    remove_edge_product = combinations_with_replacement(remove_edges
        ,2)
    remove_edge_tuple = []
    for x in remove_edge_product:
        remove_edge_tuple.append(x)
    remove_edge_final = []
    for i in range(len(remove_edge_tuple)):
        if remove_edge_tuple[i][0]!= remove_edge_tuple[i][1]:
            remove_edge_final.append(remove_edge_tuple[i])
    subgraphs = []
    edge_list = list(G.edges)
    for i in range(len(remove_edge_final)):
        temp_graph = nx.Graph(G)
        edge1 = remove_edge_final[i][0]
        edge1 = int(edge1)
        edge2 = remove_edge_final[i][1]
        edge2 = int(edge2)
        edge1_loc = edge_list[edge1]
        edge2_loc = edge_list[edge2]
        temp_graph.remove_edge(edge1_loc[0],edge1_loc[1])
        temp_graph.remove_edge(edge2_loc[0],edge2_loc[1])
        subgraphs.append(temp_graph)
    return subgraphs
# Check whether a network is empty(no edges)
def empty_check(G):
    if len(G.nodes()) == 0 or len(G.edges())==0:
        return True
    else:
        return False
# Convert a dataframe to a network
def df_to_network(df):
    source = [row[0] for index,row in df.iterrows()]
    target = [row[1] for index,row in df.iterrows()]
    G = nx.Graph()
    [G.add_edge(item1,item2) for item1,item2 in zip(source,target)]
    return G
# Finding the giant component of a network
```

```python
def gcc(G):
    Gcc = sorted(nx.connected_components(G), key=len, reverse=True)
    G0 = G.subgraph(Gcc[0])
    return G0
# Plotting the degree distribution of a graph
def plot_deg_dist(G):
    degree_sequence = sorted([d for n, d in G.degree()], reverse =
        True)
    degreeCount = collections.Counter(degree_sequence)
    deg, cnt = zip(*degreeCount.items())
    deg = list(deg)
    cnt = list(cnt)
    plt.bar(deg, cnt)
    return 0
# Convert a NetworkX graph object to a dataframe
def network_to_df(G):
    edges = list(G.edges())
    source = [item[0] for item in edges]
    target = [item[1] for item in edges]
    df = pd.DataFrame(data = {"source":source,"target":target})
    return df
# Rewire the network G with given probability prob, using pairwise
    rewiring
def pairwise_rewiring(G, prob):
    rewire_number = int(prob*len(G.edges))
    G1 = G.copy()
    c = 0
    while c != rewire_number:
        edges = list(G1.edges())
        rewire_edges = random.sample(edges,2)
        source1 = rewire_edges[0][0]
        source2 = rewire_edges[1][0]
        target1 = rewire_edges[0][1]
        target2 = rewire_edges[1][1]
        if len(set([source1, source2, target1, target2]))==4:
            if not G1.has_edge(source1, target2) and not G1.has_edge(
                source2, target1):
                G1.remove_edge(source1, target1)
                G1.remove_edge(source2, target2)
                G1.add_edge(source1, target2)
                G1.add_edge(source2, target1)
                if nx.is_connected(G1):
                    c = c+1
                else:
                    G1.add_edge(source1, target1)
                    G1.add_edge(source2, target2)
```

31

```python
                    G1.remove_edge(source1, target2)
                    G1.remove_edge(source2, target1)
    return G1
# Rewire the network G with given probability prob, using single
#     link rewiring
def single_link_rewiring(G, prob):
    G1 = G.copy()
    rewire_number = int(prob * len(G1.edges))
    for i in range(rewire_number):
        flag = 0
        while flag ==0:
            source = random.choice(list(G1.nodes()))
            neighbors = list(G1.neighbors(source))
            remove_neighbor = random.choice(neighbors)
            G1.remove_edge(source, remove_neighbor)
            if not nx.is_connected(G1):
                G1.add_edge(source, remove_neighbor)
            else:
                options = set(list(G1.nodes)) - set(list(G1.neighbors
                    (source)))-set([source, remove_neighbor])
                rewire_to = random.choice(list(options))
                G1.add_edge(source, rewire_to)
                flag = 1
    return G1
# Calculates the mutual information of a graph
def mutual_info(G):
    edges = list(G.edges)
    m = len(edges)
    I = 0
    for item in edges:
        d0 = len(list(G.neighbors(item[0])))
        d1 = len(list(G.neighbors(item[1])))
        I = I + log(2*m/(d0*d1))
    return I/m
# Calculates the redundancy of a graph
def redundancy(G):
    edges = list(G.edges)
    m = len(edges)
    R = 0
    for item in edges:
        d0 = len(list(G.neighbors(item[0])))
        d1 = len(list(G.neighbors(item[1])))
        R = R + log((d0*d1))
    return R/m
# Return the complement of a graph
def complement_graph(G):
```

```python
    edges = list(G.edges)
    nodes = list(G.nodes)
    product_list =[]
    for i in range(len(nodes)):
        for j in range(i+1,len(nodes)):
            product_list.append((nodes[i],nodes[j]))
    complement = list(set(product_list)-set(edges))
    new_G = nx.Graph()
    for item in complement:
        new_G.add_edge(item[0],item[1])
    if len(G.nodes)==len(new_G.nodes):
        if nx.is_connected(new_G):
            return new_G
    return None
# Calculate the L_r of a graph(average distance of a random graph
    with given m and n)
def lr(G):
    n = len(G.nodes)
    k = 2*len(G.edges)/n
    return log(n)/log(k)
```

C   Complexity.py

```python
import numpy as np
import networkx as nx
from math import cos,pi,log
import utilities as ut
from scipy.linalg import eig
import math
# All complexity measure have an optional parameter normalisation
# If normalisation = True(set by default), the normalised value
    will be returned. Otherwise, the unnormalized form will be
    returned

# Calculates OdC Complexity of a graph
def OdC(G, normalisation = True):
    if ut.empty_check(G) == True:
        return 0
    else:
        #Create a degree correlation matrix, using the max degree
        degree_sequence = sorted([d for n, d in G.degree()], reverse=
            True)
        max_degree = max(degree_sequence)
        degree_correlation = np.zeros((max_degree,max_degree))
```

```
        #Building the correlation matrix
        for node in list(G.nodes):
            #An array to store all the neighbors degrees
            neighbors_degree = []
            #Getting the degree of the current node
            node_degree = G.degree(node)
            #Stating all neighbors and finding their degrees
            neighbors = list(G.neighbors(node))
            neighbors_degrees_tuple=G.degree(neighbors)
            for item in neighbors_degrees_tuple:
                neighbors_degree.append(item[1])
            #For every occurence, adding one to the matrix
            for item in neighbors_degree:
                if node_degree<=item:
                    degree_correlation[node_degree-1,item-1] +=1
        #Calculating a_k
        a_k=[]
        for i in range(max_degree):
            a_k.append(sum(degree_correlation[i]))
        A = sum(a_k)
        if A !=0:
            for i in range(len(a_k)):
                a_k[i]=a_k[i]/A
        #Calculating the complexity
        complexity = 0
        for item in a_k:
            complexity -= item*ln(item)
        #Normalisation
        if normalisation == True:
            complexity = complexity/(ln(len(G.nodes)-1))
        return complexity
# Calculates ln(x), if x=0, return 0
def ln(x):
    if x == 0:
        return 0
    else:
        return np.log(x)
# Calculates Cr complexity of a graph
def Cr(G,normalisation = True):
    if ut.empty_check(G) == True:
        return 0
    else:
        if not nx.is_connected(G):
            return 0
        adj_matrix = nx.adjacency_matrix(G)
        adj_matrix = adj_matrix.todense()
```

```python
        eigenvalues, eigenvectors = np.linalg.eig(adj_matrix)
        r = max(eigenvalues)
        r = r.real
        if normalisation == True:
            n = len(G.nodes)
            c_r_numerator = r-2 * cos(pi/(n+1))
            c_r_denominator = n-1-2*cos(pi/(n+1))
            c_r = c_r_numerator/c_r_denominator
            Cr_complexity = 4*c_r*(1-c_r)
            return Cr_complexity
        else:
            return r
# Calculates Ce complexity of a graph
def Ce(G, normalisation = True):
    if not nx.is_connected(G):
        return 0
    if ut.empty_check(G):
        return -1
    else:
        E = 0
        nodes = list(G.nodes())
        n=len(nodes)
        for i in range(n):
            for j in range(i+1, n):
                E = E + 1/nx.shortest_path_length(G,nodes[i],nodes[j
                    ])
        E = 2* E /((n)*(n-1))
        if normalisation ==True:
            E_path = 0
            for i in range(1,n):
                E_path = E_path + (n-i)/i
            E_path = E_path *2/((n)*(n-1))
            Ce = 4*(E-E_path)/(1-E_path)*(1- (E-E_path)/(1-E_path))
            return Ce
        else:
            return E
# Calculates C_{1e,st} complexity of a graph
def C1est(G, normalisation = True):
    n = len(G.nodes)
    mcu = n**1.68-10
    subgraphs = ut.subgraph_one_edge_deletion(G)
    st = []
    for item in subgraphs:
        L = nx.laplacian_matrix(item).toarray()
        L = np.delete(L,0, axis=0)
        L = np.delete(L,0, axis=1)
```

```python
            _,logdet = np.linalg.slogdet(L)
            det = math.exp(logdet)
            det = int(det)
            det = round(det,10)
            if det not in st:
                st.append(det)
    N1est = len(st)
    if normalisation == False:
        return N1est
    else:
        return (N1est-1)/(mcu-1)
# Calculates C_{1e.spec} complexity of a graph
def C1espec(G, normalisation =True):
    subgraphs = ut.subgraph_one_edge_deletion(G)
    spectra = []; spectra_s = []
    mcu = len(G.nodes())**1.68-10
    for i in range(len(subgraphs)):
        L = nx.laplacian_matrix(subgraphs[i]).todense()
        A = nx.adjacency_matrix(subgraphs[i]).todense()
        eig_values,_ = eig(L)
        eig_values = eig_values.real
        eig_values = sorted(eig_values)
        eig_values = [round(item,10) for item in eig_values]
        L = L + A + A
        eig_values_s,_ = eig(L)
        eig_values_s = eig_values_s.real
        eig_values_s = sorted(eig_values_s)
        eig_values_s = [round(item,10) for item in eig_values_s]
        if eig_values not in spectra and eig_values_s not in
            spectra_s:
            spectra.append(eig_values)
            spectra_s.append(eig_values_s)
    N1espec = len(spectra)
    if normalisation == False:
        return N1espec
    else:
        return (N1espec-1)/(mcu-1)
# Calculates C_{2e,spec} complexity of a graph
def C2espec(G, normalisation=True):
    n= len(G.nodes)
    remove_edges = []
    for i in range(n):
        neighbours = list(G.neighbors(i))
        for item in neighbours:
            if item < i:
                remove_edges.append([i,item])
```

```python
        products = []
        for i in range(len(remove_edges)):
            for j in range(i+1,len(remove_edges)):
                products.append([remove_edges[i],remove_edges[j]])
        subgraphs = []
        for item in products:
            temp_G = G.copy()
            temp_G.remove_edge(item[0][0],item[0][1])
            temp_G.remove_edge(item[1][0],item[1][1])
            subgraphs.append(temp_G)
        spectra = []; spectra_s = []
        mcu = len(G.nodes())**1.68-10
        for i in range(len(subgraphs)):
            L = nx.laplacian_matrix(subgraphs[i]).todense()
            A = nx.adjacency_matrix(subgraphs[i]).todense()
            eig_values,_ = eig(L)
            eig_values = eig_values.real
            eig_values = sorted(eig_values)
            eig_values = [round(item,10) for item in eig_values]
            L = L + A + A
            eig_values_s,_ = eig(L)
            eig_values_s = eig_values_s.real
            eig_values_s = sorted(eig_values_s)
            eig_values_s = [round(item,10) for item in eig_values_s]
            if eig_values not in spectra and eig_values_s not in
                spectra_s:
                    spectra.append(eig_values)
                    spectra_s.append(eig_values_s)
        N2espec = len(spectra)
        if normalisation == False:
            return N2espec
        else:
            C2espec = (N2espec - 1)/(math.comb(int(mcu),2))
            return C2espec
#   Calculates MA_{g} of a graph
#   Using function mutual_info and redundancy from utilities to
    calcualte I and R
def MAg(G, normalisation = True):
    n = len(G.nodes)
    R = ut.redundancy(G)
    I = ut.mutual_info(G)
    R_p = 2*(n-2)/(n-1)*log(2)
    R_c = 2*log(n-1)
    I_p = log(n-1)-((n-3)/(n-1))*log(2)
    I_c = log((n)/(n-1))
    MAr = 4*((R-R_p)/(R_c - R_p))*(1 - (R-R_p)/(R_c-R_p))
```

37

```python
    MAi = 4*((I-I_c)/(I_p-I_c))*(1-(I-I_c)/(I_p-I_c))
    if normalisation == True:
        return MAr * MAi
    else:
        return R*I
# Calcualtes MA_{RI} complexity of a graph
def MAri(G, normalisation=True):
    n = len(G.nodes)
    R = ut.redundancy(G)
    I = ut.mutual_info(G)
    R_p = 2*log(2)*(n-2)/(n-1)
    R_c = 2*log(n-1)
    I_p = log(n-1)-log(2)*(n-3)/(n-1)
    I_c = log(n/(n-1))
    m=len(G.edges)
    numerator_1 = (R-R_p)
    numerator_2 = (I-I_c)
    denominator = 0.25*(log(2*m)-R_p-I_c)**2
    if normalisation == True:
        return numerator_1*numerator_2/denominator
    else:
        return R*I
```

D   modify_bus.py

```python
#Remove all nodes with degree 2 in a graph, aka modified bus
def remove_degree_2(G1):
    G= G1.copy()
    flag = 1
    while flag == 1:
        node_len = len(G.nodes)
        remove_list = []
        for node in list(G.nodes):
            if G.degree(node) == 2:
                remove_list.append(node)

        for node in remove_list:
            neighbors = tuple(G.neighbors(node))
            if len(neighbors) == 2:
                G.remove_node(node)
                G.add_edge(neighbors[0],neighbors[1])
        if node_len == len(G.nodes):
            flag =0
    return G
```

E figure_generator.py

```python
import utilities as ut
import networkx as nx
import Complexity as cx
import matplotlib.pyplot as plt
import collections
from math import log
import numpy as np
import pandas as pd
```

The following codes generate figure 2:

```python
n = 10
G1 = nx.path_graph(n)
plt.figure(figsize=(10,5))
plt.subplot(1,2,1)
nx.draw(G1)
G2=nx.gnp_random_graph(n,1)
plt.subplot(1,2,2)
nx.draw(G2)
plt.show()
plt.savefig("figures/path_clique.eps",format="eps")
```

The following codes generate figure 9:

```python
import networkx as nx
import utilities as ut
import Complexity as cx
import matplotlib.pyplot as plt
import utilities as ut
import numpy as np
import matplotlib as mpl
from math import log
import pandas as pd
import collections
```

Following codes generate figure 6:

```python
n=15
graphs, df = ut.random_networks(n, False ,1000)
c1est = [cx.C1est(g) for g in graphs]
c1espec = [cx.C1espec(g) for g in graphs]
c2espec = [cx.C2espec(g) for g in graphs]
fig, axes = plt.subplots(1, 3,figsize = ([15,5]))
axes[0].scatter(df["Number_of_edges"],c1est, s=10,color = "black")
axes[1].scatter(df["Number_of_edges"],c1espec, s=10,color = "black")
axes[2].scatter(df["Number_of_edges"],c2espec, s=10,color = "black")
```

```python
for item in axes:
    item.set_yticks([0,0.5,1,1.5])
    item.set_xlabel("m",fontsize =18)
axes[0].set_ylabel("$C1e,st$",fontsize= 18)
axes[1].set_ylabel("$C1e,spec$",fontsize= 18)
axes[2].set_ylabel("$C2e,spec$",fontsize= 18)
plt.show()
fig.savefig("figures/subgraph_measures.eps",format="eps")


# The following codes generate figure 7:

n = [6,7,8]
c2graphs= []; dfs= []
for item in n:
    g,d = ut.random_networks(item,True,50)
    c2graphs.append(g); dfs.append(d)
c2especs = []
for item in c2graphs:
    temp_result = [cx.C2espec(g) for g in item]
    c2especs.append(temp_result)
fig,axes = plt.subplots(1,3,figsize = (15,5))
for i in range(len(n)):
    axes[i].scatter(dfs[i]["Number_of_edges"],c2especs[i],s=10,color
        = "black")
    axes[i].set_yticks([0,0.5,1])
    axes[i].set_title("n="+str(n[i]),fontsize= 18)
    axes[i].set_xlabel("m",fontsize= 18)
axes[0].set_ylabel("$C_{1e,st}$",fontsize= 18)
axes[1].set_ylabel("$C_{1e,spec}$",fontsize= 18)
axes[2].set_ylabel("$C_{2e,spec}$",fontsize= 18)
plt.show()
fig.savefig("figures/c2espec.eps",format="eps")


# The following codes generate figure 4:

plt.figure(figsize =(15,10))
plt.subplot(2,3,1)
nx.draw_circular(nx.watts_strogatz_graph(20,4,0),node_size =100)
plt.title("p = 0",fontsize= 18)
plt.subplot(2,3,2)
nx.draw_circular(nx.watts_strogatz_graph(20,4,0.1),node_size =100)
plt.title("p = 0.1",fontsize= 18)
plt.subplot(2,3,3)
nx.draw_circular(nx.watts_strogatz_graph(20,4,0.5),node_size =100)
plt.title("p = 0.5",fontsize= 18)
plt.subplot(2,3,4)
```

```
nx.draw_circular(nx.newman_watts_strogatz_graph(20,4,0),node_size
    =100)
plt.title("p_=_0",fontsize= 18)
plt.subplot(2,3,5)
nx.draw_circular(nx.newman_watts_strogatz_graph(20,4,0.1),node_size
    =100)
plt.title("p_=_0.1",fontsize= 18)
plt.subplot(2,3,6)
nx.draw_circular(nx.newman_watts_strogatz_graph(20,4,0.5),node_size
    =100)
plt.title("p_=_0.5",fontsize= 18)
plt.show()
plt.savefig("figures/small_world_network_model.eps",format = "eps")


# Following codes generate figure 5:

def degree_distribution(G):
    degree_sequence = sorted([d for n, d in G.degree()],reverse =
        True)
    degreeCount = collections.Counter(degree_sequence)
    deg, cnt = zip(*degreeCount.items())
    deg = list(deg)
    cnt = list(cnt)
    return deg,cnt
plt.figure(figsize=(15,6))
plt.subplot(1,2,1)
G = nx.gnm_random_graph(1000,5000)
deg,cnt = degree_distribution(G)
cnt = [item/1000 for item in cnt]
plt.plot(deg,cnt,color = "black",label="$G(n,m)$_model")
G = nx.barabasi_albert_graph(1000,5)
deg1,cnt1 = degree_distribution(G,)
cnt1 = [item/1000 for item in cnt1]
plt.plot(deg1,cnt1,'--',color = "black",label = "Barabasi-Albert_
    model")
plt.legend(fontsize=12)
plt.scatter(deg,cnt,color = "black",marker = "x")
plt.scatter(deg1,cnt1,color = "black")
plt.xlabel("Degree",fontsize= 18)
plt.ylabel("Probability",fontsize= 18)
plt.title("Degree_distribution",fontsize= 15)
plt.subplot(1,2,2)
log_deg = [log(item) for item in deg];log_cnt = [log(item) for item
    in cnt]
log_deg1 = [log(item) for item in deg1];log_cnt1 = [log(item) for
    item in cnt1]
```

41

```python
plt.scatter(log_deg, log_cnt, marker = "x", color = "grey", s=15, label =
    "$G(n,m) model$")
plt.scatter(log_deg1, log_cnt1, marker = "o", color = "black", s=15, label
    = "Barabasi-Albert model")
plt.legend(fontsize=12)
plt.title("Log-log plot of the degree distribution")
plt.savefig("figures/degree_distribution.eps", format = "eps")


# The following codes generate figure 8:

n = 20
graphs, df = ut.random_networks(n, False, 1000)
mari = [cx.MAri(g) for g in graphs]
mag = [cx.MAg(g) for g in graphs]
plt.figure()
plt.scatter(df["Number_of_edges"], mari, label = "$MA_{RI}$", color = "
    black", marker = "o", s=10)
plt.scatter(df["Number_of_edges"], mag, label = "$MA_{g}$", color = "
    grey", marker = "x", s=10)
plt.legend(fontsize=12)
plt.xlabel("m", fontsize= 18)
plt.ylabel("Complexity", fontsize= 18)
plt.savefig("figures/mariandmag.eps", format="eps")

methods = ["C1est", "C1espec", "C2espec", "OdC", "MAg", "MAri", "Cr", "Ce"]
#Generates random graphs and data
n=7
graphs, df = ut.random_networks(n=n, use_all_m = True, sample_number =
    50)
#Find the complexities of the graphs
results = []
for item in methods:
    method = getattr(cx, item)
    temp_result = [method(g) for g in graphs]
    results.append(temp_result)
c=0
fig, axes = plt.subplots(4, 2, figsize = (10, 16))
xticks = np.linspace(n-1, n*(n-1)/2, 5)
xticks = [int(item) for item in xticks]
for i in range(4):
    for j in range(2):
        axes[i][j].scatter(df["Number_of_edges"], results[c], s=10,
            color = "black")
        axes[i][j].set_yticks([0, 0.5, 1])
        axes[i][j].set_xticks(xticks)
        axes[i][j].set_xlabel("m", fontsize = 18)
```

```python
            axes[i][j].set_ylabel("$"+methods[c]+"$",fontsize = 18)
            c+=1
axes[0][0].set_ylabel("$C_{1e,st}$",fontsize= 18)
axes[0][1].set_ylabel("$C_{1e,spec}$",fontsize= 18)
axes[1][0].set_ylabel("$C_{2e,spec}$",fontsize= 18)
plt.savefig("figures/complexities.eps",format="eps")
plt.show()
```

Following codes generate figure 12:

```python
n=25
corr_use_graphs,df = ut.random_networks(n,False,1000)
corr_results = []
corr_results.append([cx.C1est(g) for g in corr_use_graphs])
corr_results.append([cx.OdC(g) for g in corr_use_graphs])
corr_results.append([cx.MAri(g) for g in corr_use_graphs])
corr_use_BA = ut.BA_random_graphs(n,200)
corr_results_BA = []
corr_results_BA.append([cx.C1est(g) for g in corr_use_BA])
corr_results_BA.append([cx.OdC(g) for g in corr_use_BA])
corr_results_BA.append([cx.MAri(g) for g in corr_use_BA])
plt.figure(figsize=(22,7))
plt.subplot(1,3,1)
plt.scatter(corr_results[0],
                corr_results[1],
                s=15,c = df["Number_of_edges"],cmap = "binary",label
                    = "$G(n,m)$_random_graphs")
plt.scatter(corr_results_BA[0],corr_results_BA[1],color="black",
    marker = "x",label = "BA_Graphs")
plt.xlabel("$C_{1e,st}$",fontsize = 20);plt.ylabel("$OdC$",fontsize =
    20);

plt.subplot(1,3,2)
plt.scatter(corr_results[0],
                corr_results[2],
                s=15,c = df["Number_of_edges"],cmap = "binary",label
                    = "$G(n,m)$_random_graphs")
plt.scatter(corr_results_BA[0],corr_results_BA[2],color="black",
    marker = "x",label = "BA_Graphs")
plt.xlabel("$C_{1e,st}$",fontsize = 20);plt.ylabel("$MAri$",fontsize
    = 20);
plt.legend(fontsize = 15)
plt.subplot(1,3,3)
plt.scatter(corr_results[1],
                corr_results[2],
                s=15,c = df["Number_of_edges"],cmap = "binary",label
```

```
                        = "$G(n,m)$_random_graphs")
plt.colorbar()
plt.scatter(corr_results_BA[1],corr_results_BA[2],color="black",
    marker = "x",label = "BA_Graphs")
plt.xlabel("$OdC$",fontsize = 20);plt.ylabel("$MAri$",fontsize = 20);
plt.show()
plt.savefig("figures/complexity_correlation.eps",format = "eps")
```

Following codes generate figure 10:

```
#Generates special random graphs
n = 20
randoms,random_df = ut.random_networks(n,False,500)
BA_graphs = ut.BA_random_graphs(n=n,sample_number = 100)
WS_graphs = ut.WS_random_graphs(n=n,sample_number = 100)
NW_graphs = ut.NW_random_graphs(n=n,sample_number = 100)
#Calculates the complexity of special random graphs
rd_result = []
for item in methods:
    method = getattr(cx,item)
    temp_result = [method(g) for g in randoms]
    rd_result.append(temp_result)
BA_result = []
for item in methods:
    method = getattr(cx,item)
    temp_result = [method(g) for g in BA_graphs]
    BA_result.append(temp_result)
WS_result = []
for item in methods:
    method = getattr(cx,item)
    temp_result = [method(g) for g in WS_graphs]
    WS_result.append(temp_result)
NW_result = []
for item in methods:
    method = getattr(cx,item)
    temp_result = [method(g) for g in NW_graphs]
    NW_result.append(temp_result)
# Calculates the complexities of special graphs
n=20
c=0
fig,axes = plt.subplots(4,2,figsize = (10,16))
xticks = np.linspace(n-1,n*(n-1)/2,5)
xticks = [int(item) for item in xticks]
for i in range(4):
    for j in range(2):
        axes[i][j].scatter([len(g.edges) for g in randoms],rd_result[
```

```
             c ] , s =15, color  = ” black ” , alpha  =  0.7 , label  =  ”$G( n ,m) $ ” )
            axes [ i ] [ j ] . scatter ( [ len ( g . edges )  for  g  in  BA_graphs ] ,
                BA_result [ c ] , s =15, color  =  ” green ” , label  =  ”BA” )
            axes [ i ] [ j ] . scatter ( [ len ( g . edges )  for  g  in  WS_graphs ] ,
                WS_result [ c ] , marker  =  ”x” , s =15, color  =  ” red ” , label  =  ”WS” )
            axes [ i ] [ j ] . scatter ( [ len ( g . edges )  for  g  in  NW_graphs ] ,
                NW_result [ c ] , marker  =  ”s” , s =15, color  =  ” blue ” , label  =  ”NW”
                )
            axes [ i ] [ j ] . set_yticks ( [ 0 ,0.5 ,1 ] )
            axes [ i ] [ j ] . set_ylabel ( ”$”+methods [ c ]+”$” , fontsize  =  20)
            axes [ i ] [ j ] . set_xlabel ( ”m” , fontsize  =  20)
            axes [ i ] [ j ] . set_xticks ( xticks )
            c+=1
axes [ 0 ] [ 0 ] . set_ylabel ( ”$C_{1e , st }$” )
axes [ 0 ] [ 1 ] . set_ylabel ( ”$C_{1e , spec }$” )
axes [ 1 ] [ 0 ] . set_ylabel ( ”$C_{2e , spec }$” )
axes [ 0 ] [ 0 ] . legend ()
plt . legend ( fontsize  =  12)
plt . savefig ( ” figures / complexities_sp . eps ” , format=” eps ” )
plt . show ()


# The following codes generate figure 13:

#Analyse  Complement  graphs  complexity
graphs , df  =  ut . random_networks (25 , False ,1000)
BA_graphs  =  ut . BA_random_graphs (25 ,  100)
complement_graphs  =  [ ut . complement_graph ( g )  for  g  in  graphs ]
c  =  [ ]
g  =  [ ]
edges  =  [ ]
for  i  in  range ( len ( complement_graphs ) ) :
    if  complement_graphs [ i ]  !=None :
        c . append ( complement_graphs [ i ] )
        g . append ( graphs [ i ] )
        edges . append ( df [ ” Number_of_edges ” ] [ i ] )
BA_comp  =  [ ut . complement_graph ( g )  for  g  in  BA_graphs ]
ba_c  =  [ ]
ba_g  =  [ ]
for  i  in  range ( len ( BA_comp ) ) :
    if  BA_comp [ i ]  !=None :
        ba_c . append ( BA_comp [ i ] )
        ba_g . append ( BA_graphs [ i ] )
results1  =  [ cx . OdC ( item )  for  item  in  g ]
results_c  =  [ cx . OdC ( g )  for  g  in  c ]
ba_result  =  [ cx . OdC ( g )  for  g  in  ba_c ]
ba_result_c  =  [ cx . OdC ( g )  for  g  in  ba_g ]
```

```
results2 = [cx.MAri(item) for item in g]
results_c_1 = [cx.MAri(g) for g in c]
ba_result_1 = [cx.MAri(g) for g in ba_c]
ba_result_c_1 = [cx.MAri(g) for g in ba_g]
results3 = [cx.C1est(item) for item in g]
results_c_2 = [cx.C1est(g) for g in c]
ba_result_2 = [cx.C1est(g) for g in ba_c]
ba_result_c_2 = [cx.C1est(g) for g in ba_g]
plt.figure(figsize = (18,5))
plt.subplot(1,3,1)
plt.scatter(results1 ,results_c ,c = edges ,cmap="gray",label = "$G(n,m)
    $ random graphs")
plt.colorbar()
plt.scatter(ba_result ,ba_result_c ,marker = "x",label = "BA graphs")
plt.xlabel("$OdC$ of original graph",fontsize = 20)
plt.ylabel("$OdC$ of complement graph",fontsize = 20)
plt.xticks([0,0.5,1])
plt.yticks([0,0.5,1])
plt.legend(fontsize = 12)
plt.subplot(1,3,2)
plt.scatter(results2 ,results_c_1 ,c = edges ,cmap="gray",label = "$G(n,
    m)$ random graphs")
plt.colorbar()
plt.scatter(ba_result_1 ,ba_result_c_1 ,marker = "x",label = "BA graphs
    ")
plt.xlabel("$MAri$ of original graph",fontsize = 20)
plt.ylabel("$MAri$ of complement graph",fontsize = 20)
plt.xticks([0,0.5,1])
plt.yticks([0,0.5,1])
plt.subplot(1,3,3)
plt.scatter(results3 ,results_c_2 ,c = edges ,cmap="gray",label = "$G(n,
    m)$ random graphs")
plt.colorbar()
plt.scatter(ba_result_2 ,ba_result_c_2 ,marker = "x",label = "BA graphs
    ")
plt.xlabel("$C_{1e,st}$ of original graph",fontsize = 20)
plt.ylabel("$C_{1e,st}$ of complement graph",fontsize = 20)
plt.xticks([0,0.5,1])
plt.yticks([0,0.5,1])
plt.legend(fontsize = 12)
plt.savefig("figures/complement.eps",format="eps")
```

Following codes generate figure 14. Used graphs' descriptions can be found in table 1:

```
index = ["dolphins","pdzbase","hamsterster","Roget","flight","
    GBPT_train"]
```

```
index_bus = ["london","paris","berlin","sydney","detroit","beijing"]
load_path = ["real_networks/processed/"+item+".csv" for item in index
    ]
load_path_bus = ["bus/processed/"+item+".csv" for item in index_bus]
load_path_bus_m = ["bus/modified/m_"+item+".csv" for item in
    index_bus]
graphs = [ut.df_to_network(pd.read_csv(item)) for item in load_path]
bus = [ut.df_to_network(pd.read_csv(item)) for item in load_path_bus]
bus_m = [ut.df_to_network(pd.read_csv(item)) for item in
    load_path_bus_m]
l = [nx.average_shortest_path_length(g) for g in graphs]
l_bus = [nx.average_shortest_path_length(g) for g in bus]
l_bus_m = [nx.average_shortest_path_length(g) for g in bus_m]
lr = [ut.lr(g) for g in graphs]
lr_bus = [ut.lr(g) for g in bus]
lr_bus_m = [ut.lr(g) for g in bus_m]
result_g = [cx.MAri(g) for g in graphs]
result_bus = [cx.MAri(g) for g in bus]
result_bus_m = [cx.MAri(g) for g in bus_m]
generated_graphs = [
    nx.gnm_random_graph(875,4000),
    nx.watts_strogatz_graph(875,10,0.05),
    nx.newman_watts_strogatz_graph(875,9,0.05),
    nx.barabasi_albert_graph(1000,4)
]
l_gg_ratio =[]
for i in range(len(generated_graphs)):
    l_gg_ratio.append(nx.average_shortest_path_length(
        generated_graphs[i])/ut.lr(generated_graphs[i]))
gg_result = [cx.MAri(item) for item in generated_graphs]
plt.figure(figsize=(16,8))
l_ratio = [item0/item1 for item0,item1 in zip(l,lr)]
l_ratio_bus = [item0/item1 for item0,item1 in zip(l_bus,lr_bus)]
l_ratio_bus_m = [item0/item1 for item0,item1 in zip(l_bus_m,lr_bus_m)
    ]
plt.scatter(l_ratio,result_g,color = "black",s=50,label = "Real_
    networks")
plt.scatter(l_ratio_bus,result_bus,color = "red",s=50,marker = "x",
    label ="Bus_networks")
plt.scatter(l_ratio_bus_m,result_bus_m,color = "blue",marker = "^",s
    =50,label = "Modified_bus_networks")
plt.scatter(l_gg_ratio,gg_result,color = "green",marker = "s",s=50,
    label = "Generated_graphs")
plt.legend(fontsize = 15)
plt.xlabel("$L/L_r$",fontsize = 20)
plt.ylabel("$MAri$",fontsize = 20)
```

```
plt.yticks([0,0.25,0.5,0.75,1])
for i in range(len(index)):
    if i !=1:
        plt.annotate(str(i+1),(l_ratio[i]+0.02,result_g[i]+0.02),
            fontsize= 15,color = "grey")
for i in range(len(index)):
    plt.annotate(str(i+7),(l_ratio_bus[i]+0.02,result_bus[i]+0.02),
        fontsize= 15,color = "grey")
for i in range(len(index)):
    plt.annotate(str(i+13),(l_ratio_bus_m[i]+0.02,result_bus_m[i
        ]+0.02),fontsize= 15,color = "grey")
for i in range(len(generated_graphs)):
    plt.annotate(str(i+19),(l_gg_ratio[i]+0.02,gg_result[i]+0.02),
        fontsize= 15,color = "grey")
plt.annotate("2",(1,0.275),fontsize=15,color="grey")
plt.savefig("figures/real_networks.eps",format = "eps")
```

F   rewiring_simulation.py

This file is used to simulate both rewirings introduced in section 1.3, and generate figure 15:

```
import utilities as ut
import Complexity as cx
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
class rewiring():
    def __init__(self,name,bus=False,step_prob = 0.05):
        self.name = name
        self.bus = bus
        self.load_network()
        self.step_prob = step_prob
        self.rewired_G = self.G.copy()
        self.result = []
        self.s_n=0
        self.p_n=0
        self.load_result("S")
        self.load_result("P")
    #Load all the networks
    def load_network(self):
        if self.bus == False:
            load_path = "real_networks/processed/"+self.name+".csv"
        else:
            load_path = "bus/modified/m_"+self.name+".csv"
        df = pd.read_csv(load_path)
```

```python
        self.G = ut.df_to_network(df)
    #Carry out one step of single link rewiring
    def single_link_rewiring(self):
        self.rewired_G = ut.single_link_rewiring(self.rewired_G, self.
            step_prob)
    #Carry out one step of pairwise rewiring
    def pairwise_rewiring(self):
        self.rewired_G = ut.pairwise_rewiring(self.rewired_G, self.
            step_prob)
    #Reinstantiate the rewired graph
    def clear_rewiring(self):
        self.rewired_G = self.G.copy()
        if cx.OdC(self.rewired_G) != cx.OdC(self.G):
            Exception("Clearing failed")
    #Carry out a sample number of rewiring
    def continuous_rewiring(self, method):
        self.result = [cx.OdC(self.G)]
        if method == "S":
            niter = int(1/self.step_prob)
            for i in range(niter):
                self.single_link_rewiring()
                self.result.append(cx.OdC(self.rewired_G))
            self.s_n += 1
        elif method == "P":
            niter = int(1/self.step_prob)
            for i in range(niter):
                self.pairwise_rewiring()
                self.result.append(cx.OdC(self.rewired_G))
            self.p_n += 1
        else:
            raise ValueError('Insufficient method')
    #Load the calculated result from the data folder
    def load_result(self, method):
        if method =="S":
            try:
                self.s_result = pd.read_csv("result/single_link/"+
                    self.name+".csv")
                self.s_n = len(self.s_result.columns)
            except FileNotFoundError:
                self.s_result = pd.DataFrame(data = {"1":[]})
                self.s_n = 0
        if method =="P":
            try:
                self.p_result = pd.read_csv("result/pairwise/"+self.
                    name+".csv")
                self.p_n = len(self.p_result.columns)
```

49

```python
        except FileNotFoundError:
            self.p_result = pd.DataFrame(data = {"1":[]})
            self.p_n = 0
#Store the result in the class variable result
def record_result(self, method):
    if method == "S":
        self.s_result[str(self.s_n)] = self.result
    elif method == "P":
        self.p_result[str(self.p_n)] = self.result
    else:
        raise ValueError('Insufficient method')
#Save the current result table as a csv file
def write_result(self, method):
    if method == "S" and self.s_n == 0:
        raise Exception("There is no recorded result")
    if method == "P" and self.p_n == 0:
        raise Exception("There is no recorded result")
    if (method == "S+P" or method == "P+S") and (self.s_n == 0 or
        self.p_n==0):
        raise Exception("There is no recorded result for one/both
            of the methods.")
    if method == "S":
        self.s_result.to_csv("result/single_link/"+self.name+".
            csv",index = False)
    elif method == "P":
        self.p_result.to_csv("result/pairwise/"+self.name+".csv",
            index = False)
    elif method == "S+P" or method == "P+S":
        self.s_result.to_csv("result/single_link/"+self.name+".
            csv",index = False)
        self.p_result.to_csv("result/pairwise/"+self.name+".csv",
            index = False)
    else:
        raise ValueError('Insufficient method')
#Carry multiple number of rewiring according to the given
    parameter
def experiment(self, method, sample_number):
    if type(sample_number) != type(int(1)):
        ValueError("Insufficient number of samples, please input
            an integer.")
    for i in range(sample_number):
        self.clear_rewiring()
        self.continuous_rewiring(method)
        self.record_result(method)
#Plot the results
def plot_result(self, method):
```

```python
if method == "S":
    if self.s_n != 0:
        prob_list = np.linspace(0,1,int(1/(self.step_prob)+1)
            )
        mean = [np.mean(row) for index, row in self.s_result.
            iterrows()]
        std = [0.5*np.std(row) for index, row in self.
            s_result.iterrows()]
        plt.errorbar(prob_list,mean,yerr = std,color = "black
            ")
    else:
        raise Exception("There is no recorded result")
elif method == "P":
    if self.p_n != 0:
        prob_list = np.linspace(0,1,int(1/(self.step_prob)+1)
            )
        mean = [np.mean(row) for index, row in self.p_result.
            iterrows()]
        std = [0.5*np.std(row) for index, row in self.
            p_result.iterrows()]
        plt.errorbar(prob_list,mean,yerr = std,color = "black
            ")
    else:
        raise Exception("There is no recorded result")
elif method == "P+S" or method == "S+P":
    if self.s_n !=0 and self.p_n != 0:
        prob_list = np.linspace(0,1,int(1/(self.step_prob)+1)
            )
        mean = [np.mean(row) for index, row in self.s_result.
            iterrows()]
        std = [0.5*np.std(row) for index, row in self.
            s_result.iterrows()]
        plt.errorbar(prob_list,mean,yerr = std,color = "black
            ",label = "Single link rewiring")
        mean = [np.mean(row) for index, row in self.p_result.
            iterrows()]
        std = [0.5*np.std(row) for index, row in self.
            p_result.iterrows()]
        plt.errorbar(prob_list,mean,yerr = std,color = "blue"
            ,label = "Pairwise rewiring")
        plt.legend()
        plt.xlabel("Rewiring Probability")
        plt.ylabel("OdC Complexity")
        plt.title("Change of complexity after rewiring")
    else:
```

```python
                raise Exception("There␣is␣no␣recorded␣result␣for␣one/
                    both␣of␣the␣methods.")
        else:
            raise ValueError('Insufficient␣method')
    #Carry out a given number of rewiring, saving the result and
        plotting at the same time.
    def one_stop(self,method,sample_number):
        self.experiment(method,sample_number)
        self.write_result(method)
        self.plot_result(method)
index = ["dolphins","pdzbase","GBPT_train",
        "hamsterster","Roget","flight"]
index_bus = ["london","paris","berlin","sydney","detroit","beijing"]
real_networks = [rewiring(item) for item in index]
bus_networks = [rewiring(item,bus = True) for item in index_bus]
for niter in range(10):
    for item in real_networks:
        item.experiment("P",1)
        item.write_result("P")
        item.experiment("S",1)
        item.write_result("S")
    for item in bus_networks:
        item.experiment("P",1)
        item.write_result("P")
        item.experiment("S",1)
        item.write_result("S")
plt.figure(figsize=(20,20))
step_prob = 0.05
prob_list = np.linspace(0,1,int(1/(step_prob)+1))
for i in range(len(real_networks)):
    plt.subplot(4,3,i+1)
    mean = [np.mean(row) for index, row in real_networks[i].s_result.
        iterrows()]
    std = [np.std(row) for index, row in real_networks[i].s_result.
        iterrows()]
    plt.errorbar(prob_list,mean,yerr = std,color = "black",label = "
        Single␣link␣rewiring")
    mean = [np.mean(row) for index, row in real_networks[i].p_result.
        iterrows()]
    std = [np.std(row) for index, row in real_networks[i].p_result.
        iterrows()]
    plt.errorbar(prob_list,mean,yerr = std,color = "blue",label = "
        Pairwise␣rewiring")
    plt.title(index[i].capitalize(),fontsize=15)
for i in range(len(real_networks)):
    plt.subplot(4,3,6+i+1)
```

```
    mean = [ np . mean ( row )  for  index ,  row  in  bus_networks [ i ] . s_result .
        iterrows ( ) ]
    std = [ np . std ( row )  for  index ,  row  in  bus_networks [ i ] . s_result .
        iterrows ( ) ]
    plt . errorbar ( prob_list , mean , yerr = std , color = " black " , label = "
        Single _ link _ rewiring " )
    mean = [ np . mean ( row )  for  index ,  row  in  bus_networks [ i ] . p_result .
        iterrows ( ) ]
    std = [ np . std ( row )  for  index ,  row  in  bus_networks [ i ] . p_result .
        iterrows ( ) ]
    plt . errorbar ( prob_list , mean , yerr = std , color = " blue " , label = "
        Pairwise _ rewiring " )
    plt . title ( index_bus [ i ] . capitalize ( ) , fontsize =15)
plt . subplot ( 4 , 3 , 1 ) ; plt . ylabel ( " $OdC$ " , fontsize =18) ; plt . legend (
    fontsize  =  12 ) ; plt . title ( " Dolphins " , fontsize =15)
plt . subplot ( 4 , 3 , 2 ) ; plt . title ( " PDZBase " , fontsize =15)
plt . subplot ( 4 , 3 , 3 ) ; plt . title ( " UK_train " , fontsize =15)
plt . subplot ( 4 , 3 , 4 ) ; plt . ylabel ( " $OdC$ " , fontsize =18) ;
plt . subplot ( 4 , 3 , 5 ) ; plt . title ( " Roget ' s _ thesaurus " , fontsize =15)
plt . subplot ( 4 , 3 , 7 ) ; plt . ylabel ( " $OdC$ " , fontsize =18)
plt . subplot ( 4 , 3 , 10 ) ; plt . xlabel ( " Rewiring _ probability " , fontsize =18) ;
    plt . ylabel ( " $OdC$ " , fontsize =18)
plt . subplot ( 4 , 3 , 11 ) ; plt . xlabel ( " Rewiring _ probability " , fontsize =18)
plt . subplot ( 4 , 3 , 12 ) ; plt . xlabel ( " Rewiring _ probability " , fontsize =18) ;
    plt . legend ( fontsize  =  12 )
plt . savefig ( " figures / rewiring . eps " , format  =  " eps " )
```

G   configuration_model.r and R_functions.py

Since no package can implement a connected configuration model in Python, R is used to implement the configuration model and simulate the behaviour to produce figure 11.
configuration_model.r:

```
#include necessary libraries , and setting up the environment
library ( " reticulate " )
library ( " igraph " )
library ( " plotly " )
use_python ( " / user / python3 " )
#function1 return the edges list to build the network in networkx
#function2 return the graph itself for tespecing in the R environment
configuration_model <- function ( series ) {
  g <- sample_degseq ( series ,  method=" vl " )
  return ( as_edgelist ( g ) )
}
return_graph<- function ( series ) {
```

```
    g <- sample_degseq(series, method="vl")
    return(g)
}
#python functions
source_python("R_functions.py")
#Building the network in python
n = 50
gamma = 3
gamma_list = seq(gamma-0.3,gamma+0.3,0.01)
mari_results = integer(length(gamma_list))
OdC_results = integer(length(gamma_list))
C1espec_results = integer(length(gamma_list))
graphs = list()
for(niter in 1:50){
for(i in 1:length(gamma_list)){
    series = power_law_series(n,gamma_list[i])
    if(sum(series)%%2 !=0){
        series[1]=series[1]+1
    }
    edge_list = configuration_model(series)
    g = construct_network(edge_list)
    graphs = append(graphs,g)
    mari_results[i] = mari_results[i]+MAri(g)
    OdC_results[i] = OdC_results[i]+OdC(g)
    C1espec_results[i] = C1espec_results[i]+C1espec(g)
}
    }
mari_results = mari_results/niter
OdC_results = OdC_results/niter
C1espec_results = C1espec_results/niter
dev.new(3,5)
plot(gamma_list, C1espec_results, col="black",pch="*","l", lty=1,
    ylim=c(0,0.5), ylab="Complexity",xlab="Gamma" )
lines(gamma_list, OdC_results, col="blue",lty=2)
lines(gamma_list, mari_results, col="red", lty=3)
legend(3.1,0.45,legend=c(expression(paste('C'["1e,spec"])),"OdC","
    MAri"),col=c("black","blue","red"),lty=c(1,2,3), ncol=1)


R_functions.py:

#Complexity measure of C_{1e,spec}, OdC and MA_{RI} can be found in
    Complexity.py, please copy and paste them here.
def ln(x):
    if x == 0:
        return 0
    else:
```

```
        return np.log(x)
def power_law_series(n,gamma):
    n = int(n)
    sequence = nx.random_powerlaw_tree_sequence(n, gamma,tries
        =100000)
    return sequence
def construct_network(edge_list):
    G=nx.Graph()
    for item in edge_list:
        G.add_edge(item[0],item[1])
    return G
```

H   References

[1]   S. Blumsack et al. "Defining power network zones from measures of electrical distance". In: 2009 IEEE Power Energy Society General Meeting. 2009, pp. 1–8.

[2]   Robert Stefko, Peter Dorcak, and František Pollák. "Virtual social networks and their utilization for promotion". In: Polish Journal of Management Studies 4 (2011), pp. 126–134.

[3]   Jennifer A Dunne, Richard J Williams, and Neo D Martinez. "Food-web structure and network theory: the role of connectance and size". In: Proceedings of the National Academy of Sciences 99.20 (2002), pp. 12917–12922.

[4]   Transport of London. Key routes in central London. https://tfl.gov.uk/maps/bus. Date Accessed 29/09/2021.

[5]   ULRIK BRANDES et al. "What is network science?" In: Network Science 1.1 (2013), pp. 1–15.

[6]   Erdős Renyi. "On random graph". In: Publicationes Mathematicate 6 (1959), pp. 290–297.

[7]   A.L. Barabási. Network Science. Cambridge University Press, 2016. ISBN: 9781107076266.

[8]   Jinho Kim et al. "Network rewiring is an important mechanism of gene essentiality change". In: Scientific Reports 2.1 (2012).

[9]   Stanley Milgram. "The small world problem". In: Psychology today 2.1 (1967), pp. 60–67.

[10]   Duncan J. Watts and Steven H. Strogatz. "Collective dynamics of 'small-world' networks". In: Nature 393.6684 (1998), pp. 440–442.

[11]   M.E.J. Newman and D.J. Watts. "Renormalization group analysis of the small-world network model". In: Physics Letters A 263.4 (1999), pp. 341–346.

[12]   Petter Holme. "Rare and everywhere: Perspectives on scale-free networks". In: Nature communications 10.1 (2019), pp. 1–3.

[13]   Anna D. Broido and Aaron Clauset. "Scale-free networks are rare". In: Nature Communications 10.1 (2019).

[14]   Réka Albert, Hawoong Jeong, and Albert-László Barabási. "Diameter of the world-wide web". In: Nature 401.6749 (1999), pp. 130–131.

[15] Thomas House et al. "Testing the hypothesis of preferential attachment in social network formation". In: EPJ Data Science 4.1 (2015), pp. 1–13.

[16] Jongkwang Kim and Thomas Wilhelm. "What is a complex graph?" In: Physica A: Statistical Mechanics and its Applications 387.11 (2008), pp. 2637–2652.

[17] Jens Christian Claussen. "Offdiagonal complexity: A computationally quick complexity measure for graphs and networks". In: Physica A 375.1 (2007), pp. 365–373.

[18] Vyacheslav B Priezzhev. "The dimer problem and the Kirchhoff theorem". In: Soviet Physics Uspekhi 28.12 (1985), p. 1125.

[19] 15. floating point ARITHMETIC: Issues and Limitations. `https://docs.python.org/3.8/tutorial/floatingpoint.html`. Date Accessed: 01/09/2021.

[20] Built-in Types. `https://docs.python.org/3/library/stdtypes.html`. Date Accessed: 15/09/2021.

[21] Thomas Wilhelm and Jens Hollunder. "Information theoretic description of networks". In: Physica A: Statistical Mechanics and its Applications 385.1 (2007), pp. 385–396.

[22] Vito Latora and Massimo Marchiori. "Economic small-world behavior in weighted networks". In: The European Physical Journal B-Condensed Matter and Complex Systems 32.2 (2003), pp. 249–263.

[23] Andries E Brouwer and Willem H Haemers. Spectra of graphs. Springer Science & Business Media, 2011.

[24] Mark Newman. Networks: An Introduction. Oxford University Press Inc., 2010. ISBN: 9780199206650.

[25] Dolphins. `http://konect.cc/networks/dolphins/`. Date Accessed: 20/09/2021.

[26] PDZBase protein-protein interaction network. `http://konect.cc/networks/maayan-pdzbase/`. Date Accessed: 20/09/2021.

[27] Hamsterster online socil network. `http://konect.cc/networks/petster-hamster-household/`. Date Accessed: 20/09/2021.

[28] Roget's Theasures. `http://vlado.fmf.uni-lj.si/pub/networks/data/dic/roget/Roget.htm`. Date Accessed: 20/09/2021.

[29] Flight network. `https://openflights.org/data.html`. Date Accessed: 20/09/2021.

[30] Riccardo Gallotti and Marc Barthelemy. "The multilayer temporal network of public transport in Great Britain". In: Scientific data 2.1 (2015), pp. 1–8.

[31] Rainer Kujala et al. "A collection of public transport network data sets for 25 cities". In: Scientific data 5.1 (2018), pp. 1–14.

[32] Beijing bus network. `https://www.mot.gov.cn/sjkf/202005/t20200528_3333174.html`. Date Accessed: 20/09/2021.

[33] Tang Zhixing, Huang Shan, and Han Songchen. "Recent Progress about Flight Delay under Complex Network". In: Complexity 2021 (2021).

[34] Frank Emmert-Streib and Matthias Dehmer. "Exploring Statistical and Population Aspects of Network Complexity". In: PLoS ONE 7.5 (2012), e34523.

[35]   Matthias Dehmer et al. "A Large Scale Analysis of Information-Theoretic Network Complexity Measures Using Chemical Structures". In: PLoS ONE 4.12 (2009), e8057.