UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA BIOMEDICA

"SIMULAZIONE DI GRAFI A TOPOLOGIA MODULARE: LA SFIDA DELLE RETI BIOLOGICHE"

Relatore: Prof. Barbara Di Camillo

Laureando: Riccardo Zanatta

ANNO ACCADEMICO 2024 – 2025

Data di laurea 19/11/2024

Sommario

0. INTRODUZIONE 2
1. PROPRIETÀ DI UN NETWORK BIOLOGICO E PRIMO APPROCCIO AL PROBLEMA 3
1.1 PROPRIETÀ TOPOLOGICHE DI UN NETWORK BIOLOGICO 3
1.2 MODELLI DI NETWORK TRASCRIZIONALI 5
1.3 GENERAZIONE DI UN MODELLO MTM 7
2. IMPLEMENTAZIONE DEL CODICE E FUNZIONAMENTO 9
2.1 GENERAZIONE DEI MODULI 9
2.1.1 MODULI E CREAZIONE DELLA LORO STRUTTURA DI BASE 9
2.1.2 FUNZIONI AUSILIARIE ALLA GENERAZIONE DELLA STRUTTURA FINALE DEL MODULO 13
2.1.3 GENERAZIONE DELLA STRUTTURA FINALE DEI MODULI 20
2.2 CALCOLO DELLO SCORE MODULARE E SELEZIONE DI UN MODULO DA UTILIZZARE 23
2.2.1 CALCOLO DELLO SCORE 23
2.2.2 SELEZIONE DI UN MODULO DA AGGIUNGERE 31
2.3 FORMAZIONE DELLA RETE E CALCOLO DEI RISULTATI 33
2.3.1 FORMAZIONE DELLA RETE 33
2.3.2 CALCOLO DEI RISULTATI 34
3. RISULTATI 37
3.1 CALCOLO DELL'ERRORE 37
3.2 CONSIDERAZIONI SUI TEMPI DI ESECUZIONE 37
3.3 TEST ESEGUITI E RISULTATI OTTENUTI 38
4. CONCLUSIONE 42
5. BIBLIOGRAFIA 43

0. INTRODUZIONE

I network biologici sono strutture presenti negli organismi viventi che svolgono funzioni essenziali per la vita. Principalmente ne esistono tre tipologie: metabolici, di interazione proteica e di regolazione trascrizionale (trascrizionali). Un network metabolico è l'insieme completo delle reazioni chimiche del metabolismo e delle interazioni regolatorie che guidano queste reazioni nella cellula. Una rete di interazione proteica è data dall'insieme delle possibili interazioni tra proteine, che possono essere di diversa natura, ovvero: possono interagire per formare un complesso proteico; una proteina può trasportane un'altra; una proteina può interagire con un'altra per modificarla. Infine, i network trascrizionali descrivono quali proteine regolano la trascrizione genica. Questo lavoro si concentrerà quasi esclusivamente sui network dell'ultima tipologia, quelli trascrizionali.

Si utilizzano gli studi[1,2], nei quali viene presentato un modello che permette la simulazione di network trascrizionali e viene scritto il codice in linguaggio Python che genera la topologia dei grafi associati. Nel modello i network trascrizionali generati sono riprodotti da dei grafi con direzione e l'espressione a RNA approssima l'espressione proteica. I nodi dei grafi rappresentano i geni e i loro sottoprodotti (RNA trascritti, RNA modificati a livello post-trascrizionale, proteine e proteine modificate a livello post-trascrizionale), mentre gli archi riproducono l'azione regolatoria tra i nodi. La simulazione di network biologici è utile per testare le proprietà di un network specifico o per analizzare come alcune proprietà dipendano dai parametri del network. Questo approccio è anche utile nel contesto del reverse-engineering, perché spesso la valutazione dell'efficacia di tali algoritmi viene eseguita su network simulati e non reali. In questo lavoro il primo capitolo presenta una descrizione delle proprietà dei network biologici considerati (trascrizionali) ed un primo approccio al problema della generazione dei grafi. Nel secondo capitolo viene analizzato il codice che genera la topologia del network e nell'ultimo sono presentati i risultati ottenuti dall'esecuzione del codice.

1. PROPRIETÀ DI UN NETWORK BIOLOGICO E PRIMO APPROCCIO AL PROBLEMA

La simulazione di un network trascrizionale deve rispettare alcune caratteristiche in modo tale da rappresentare il modello nella maniera il più vicino possibile alla realtà, rendendolo così utilizzabile in maniera efficace. I network trascrizionali possiedono delle proprietà topologiche fondamentali e la generazione dei grafi è stata studiata in modo tale da raggiungere i requisiti necessari per rappresentare queste caratteristiche.

1.1 PROPRIETÀ TOPOLOGICHE DI UN NETWORK BIOLOGICO

La prima proprietà importante in un network biologico è la distribuzione del grado di connessione dei nodi nel grafo. Si distinguono due tipologie di connessioni su un nodo: le connessioni out-degree e in-degree. Presi due nodi 'a' e 'b', se la direzione della connessione va da 'a' a 'b' si dice che il nodo 'a' possiede una connessione out-degree, mentre il nodo 'b' una in-degree. Per quanto riguarda la distribuzione del grado di connessione, sia in-degree che out-degree, viene utilizzato un modello in cui la probabilità di ogni nodo di avere K connessioni è proporzionale a una legge power-law:

$$P(K) \propto \frac{1}{K^{\gamma}}$$

Eq.1: legge power-law che rappresenta la distribuzione del grado di connessione in e out-degree in un grafo

In Eq.1 γ è un parametro costante che caratterizza la distribuzione. Questo tipo di organizzazione delle connessioni nel network è definita organizzazione scale-free. È stato dimostrato che i network metabolici[3] e possibilmente anche i trascrizionali[4], presentano questa proprietà, o per lo meno l'organizzazione scale-free ne rappresenta una buona approssimazione. In particolare, i network metabolici mostrano distribuzione del grado di connessione aderente alla power-law per entrambe la in-degree e la out-degree; per i network trascrizionali invece, la out-degree è in genere di questo tipo ma la in-degree non sempre, per alcuni organismi ha andamento esponenziale[5]. Il valore medio di γ osservato nei network metabolici di diversi organismi è uguale a 2.2[3,6]; per i trascrizionali invece ci sono osservazioni dirette sugli organismi 'Saccharomyces cerevisiae'[7] e 'Escherichia coli'[8] che mostrano γ uguale a 2.5 in entrabi i casi.

La seconda proprietà essenziale che caratterizza un network biologico è il coefficiente di clustering medio, ovvero la tendenza media tra i vicini di un nodo del network ad avere connessioni tra di loro. I vicini di un nodo sono tutti quei nodi a lui collegati, con connessioni in-degree o out-degree. Il valore di C è zero quando non esistono connessioni tra i vicini, uno se sono tutti connessi tra loro. Il coefficiente di clustering viene calcolato per ogni nodo del network e per ottenere il valore medio C viene calcolata la media tra i risultati trovati in tutti gli N nodi del grafo.

$$C_i = \frac{2q_i}{K_i \cdot (K_i - 1)}$$

Eq.2: calcolo del coefficiente di clustering per un nodo i-esimo del grafo

La formula per il calcolo del coefficiente in un nodo è Eq.2, in cui K_i è il grado di connessione del nodo i-esimo (senza distinzione tra in e out-degree), mentre q_i è il numero di connessioni presenti tra i vicini del nodo i-esimo. Nel caso in cui $K_i = 0$ o $K_i = 1$, viene approssimato $C_i = 0$. Nei network trascrizionali C ha valore vicino a 0.3, mentre per network trascrizionali il coefficiente è più basso. Una caratteristica importante di questo parametro è che il suo valore è indipendente dal numero di nodi presenti nel network[9].

La terza proprietà topologica dei network trascrizionali è legata alla distanza media di connessione tra coppie di nodi, definita "characteristic path lenth" (cpl). La distanza tra un nodo 'a' e un nodo 'b' è data dal numero minimo di connessioni tra nodi del grafo che devono essere percorse per arrivare da 'a' a 'b' o viceversa. È stato osservato[6,10] che la cpl dei network degli organismi 'Caenorhabditis elegans' e 'Escherichia coli' è minore di cinque, una proprietà topologica del network definita "small world": anche se il network contiene molti nodi, la distanza media tra le coppie di punti è un valore basso. In alcune analisi su 43 organismi[3] è stato osservato che la cpl è circa 3.2.

Una quarta proprietà importante è la modularità. Un modulo è un gruppo di nodi interconnessi tra loro che funziona all'interno del network come un'unità a sé stante. I nodi comunicano tra loro interagendo l'uno con l'altro. Si dice che un network presenta modularità quando l'organizzazione al suo interno è suddivisa tra più moduli di diverso tipo. Come dimostrato da alcuni studi, i network degli organismi biologici molto spesso presentano organizzazione modulare. Uno tra questi[14], ha analizzato i network metabolici di 75 organismi differenti e in tutti quanti è stato trovato alto grado di modularità. Le strutture modulari all'interno di un network biologico sono molteplici e hanno logica di connessione di vario tipo. Numerosi studi[7,8,15,16] hanno identificato delle tipologie modulari presenti in rilevanti quantità in diversi organismi. Questi lavori permettono di dimostrare nuovamente la rilevanza della modularità nei network biologici, ma soprattutto denotano come alcune strutture di interconnessione in natura si ripetano in organismi differenti, andando così a presupporre anche un processo di evoluzione ed ereditarietà[17] dei moduli presenti nei network di diversi esseri viventi.

L'obiettivo di questo lavoro sarà quindi quello di scrivere codice che permetta di generare dei grafi che rispettino nel miglior modo possibile le proprietà topologiche essenziali dei network trascrizionali appena descritte. I grafi dovranno quindi avere: distribuzione del grado di connessione in e out-degree proporzionali alla power law ad Eq.1, un valore di C costante al variare di N, comportamento small world e presentare modularità.

1.2 MODELLI DI NETWORK TRASCRIZIONALI

Nel corso degli anni le ricerche riguardanti la generazione di grafi che simulano network trascrizionali hanno affrontato lo studio di diverse strutture organizzative. Si procede ora con una breve analisi delle ultime per capire come mai la soluzione stabilita dai due studi presi in considerazione[1,2] (organizzazione modulare gerarchica) è la migliore tra le varie opzioni.

Inizialmente sono stati utilizzati dei modelli randomici[11,12], in cui ogni nodo ha la stessa probabilità di essere connesso a qualsiasi altro nodo nel network. Questo tipo di organizzazione non rispecchia un network trascrizionale, perché la natura delle connessioni non permette un andamento del grado di connessione proporzionale alla power-law e non è possibile la generazione di strutture modulari. È stato osservato che i network randomici hanno comportamento small world e che C decresce all'aumentare di N[9].

Successivamente sono stati introdotti i modelli scale-free[13], in modo tale che la probabilità per ogni nodo di avere un numero K di connessioni seguisse una distribuzione power-law. Anche in questo caso però la modularità è assente e C diminuisce con la crescita di N[9].

Un'alternativa a queste due organizzazioni sono i grafi geometrici[14], perché in essi è stato osservato che C rimane costante. I network geometrici sono determinati da un numero di nodi e da un raggio 'r'. Due nodi sono connessi se la loro distanza nello spazio è minore o uguale al valore del raggio. Grazie a questa modalità di connessione tra nodi, nei network geometrici si formano delle strutture modulari. Nodi vicini tra loro si organizzano in più strutture poi connesse. Tuttavia, i network geometrici non presentano distribuzione power-law del grado di connessione e hanno comportamento small world solo se 'r' supera una soglia che dipende da N.

Dato che queste tre modalità non permettono di generare grafi adeguati a descrivere network traqscrizionali, è stato proposto un modello di organizzazione gerarchica[9]. In esso è presente una singola struttura modulare ripetuta che raggruppa i nodi in vari moduli e contiene un nodo regolatore. Tutti questi nodi regolatori sono nuovamente interconnessi tra loro seguendo la struttura modulare di base fino a che non resta un singolo nodo al di sopra di tutti gli altri, generando così una struttura gerarchica. Questo modello ha distribuzione del grado di connessione proporzionale alla power-law, C costante al variare di N e comportamento small-world, però utilizzando una singola struttura modulare non ha modularità e anche se si avvicina ad un network trascrizionale sotto quasi tutte le caratteristiche non lo può comunque descrivere a pieno.

Nel modello proposto negli studi analizzati[1,2], vengono risolti i problemi dell'organizzazione precedente. Utilizzando un modello a topologia modulare gerarchica (MTM), vengono ottenute le tre caratteristiche fondamentali e allo stesso tempo sono riprodotte diverse strutture (motifs) caratteristiche di network reali. La generazione adopera moduli diversi che variano ad ogni

generazione, in quanto vengono utilizzate tre strutture di base e ogni volta creati nuovi moduli basati su di esse. Il modello permette anche di generare il grafo impostando la distribuzione del grado di connessione per valori diversi di γ (Eq.1) e di C, dando così la possibilità di creare strutture con parametri diversi. Inoltre, il modello MTM apporta anche miglioramenti al di fuori della topologia del grafo, per quanto riguarda la regolazione dell'espressione genica, argomento che però non verrà discusso in questo lavoro.

	Randomico	Scale-free	Geometrico	Gerarchico	MTM
Grado di connessione proporzionale alla power-law	No	Sì	No	Sì	Sì
C costante con il numero di nodi	No	No	Sì	Sì	Sì
Comportamento small world	Sì	Sì	Non sempre	Sì	Sì
Modularità	No	No	Sì	No	Sì

Tab.1: riassunto delle proprietà topologiche dei network nei cinque casi analizzati

1.3 GENERAZIONE DI UN MODELLO MTM

Osservato che il modello MTM è il più adatto a rappresentare dei network trascrizionali, si procede con una breve descrizione del funzionamento generale del codice che permette la creazione dei grafi. Il programma che genera i network utilizza una funzione principale, 'SimBioNeT'[2], che permette la connessione dei nodi utilizzando delle strutture modulari. Questa funzione è descritta dallo pseudo-codice ad Imm.1.

```
Input: N, Cref, gamma
Output: G

 1: G = matrice vuota N*N
 2: V = nodi di G
 3: H = Ø
 4: fino a che V ≠ Ø e H con più di un nodo:
 5:      fino a che V ≠ Ø:
 6:             M = sample(G,V,Cref,gamma)
 7:             G = connect(M)
 8:             H = H + nodi regolatori di M
 9:             V = V - nodi di M
10:      V = H
11:      H = Ø
12: return G
```

Imm.1: pseudocodice che descrive il funzionamento di 'SimBioNeT', funzione principale dell'algoritmo

Utilizzando come input il numero di nodi 'N', il coefficiente di clustering di riferimento 'Cref' e il parametro 'gamma' della legge power-law, 'SimBioNeT' genera un grafo che presenta le caratteristiche tipiche di un network trascrizionale. Quindi ha C≈Cref, grado di connettività coerente con la power-law con parametro 'gamma' e comportamento small world.

Alla linea 1 dello pseudo-codice viene inizializzato un vettore 'V' che contiene tutti i nodi di G, ovvero quei nodi che dovranno essere connessi all'iterazione attuale del ciclo a linea 5. A linea 2 viene inizializzato un vettore 'H' vuoto, che dovrà contenere i nodi da collegare all'iterazione successiva. A linea 5, fino a che 'V' non risulta vuoto, 'SimBioNeT' esegue la funzione 'sample' che seleziona un modulo da aggiungere al network secondo alcuni criteri che verranno analizzati in seguito. Poi a linea 6 alcuni nodi di 'V' vengono connessi tra loro andando a ricreare la struttura modulare selezionata.

All'interno dei moduli sono presenti dei nodi detti regolatori (definiti con precisione al capitolo 2.1.1) e hanno maggiore importanza all'interno del modulo. Tali nodi vengono aggiunti ad 'H' a linea 8 e verranno quindi connessi all'iterazione successiva del ciclo. A linea 9 vengono rimossi da 'V' i nodi che sono appena stati connessi nel grafo e l'iterazione del ciclo si conclude.

Alle linee 10 e 11, vengono assegnati gli elementi di 'H' al vettore 'V' e 'H' viene svuotato. In questa maniera 'V' conterrà i nodi da connettere nella nuova iterazione che sta per cominciare e 'H' è re-inizializzato come a linea 3. Il vettore 'V' continuerà a svuotarsi e a riempirsi nuovamente fino a che

non resta un solo nodo in ‘H’ e ‘V’ è vuoto. A quel punto non possono più generarsi connessioni, il network è costruito nella sua interezza e tutti i nodi del grafo sono interconnessi tra loro.

Una volta descritto a grandi linee il funzionamento del programma, si passa all’analisi del codice Python che permette la generazione dei grafi e si osserva la costruzione del network in tutti i suoi passaggi.

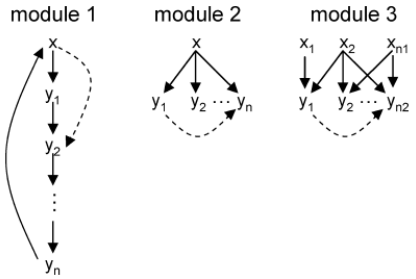
2. IMPLEMENTAZIONE DEL CODICE E FUNZIONAMENTO

Si segue il programma nel suo funzionamento, analizzando passo per passo la formazione del network per comprendere a pieno il processo di generazione del grafo. L'esecuzione del codice può essere suddivisa in tre fasi, precisamente: generazione dei moduli, calcolo dello score e formazione della rete. Si procede ora con l'analisi della prima di queste tre fasi.

2.1 GENERAZIONE DEI MODULI

Per cominciare ad analizzare il processo di formazione del grafo è necessario partire dalla generazione dei moduli, elementi fondamentali che costituiscono tutto il network. La simulazione utilizza tre tipologie di strutture modulari che hanno una logica di base fissata. Il programma genera i moduli seguendo questi concetti e poi li modifica in modo casuale per cercare di aderire ai parametri fissati ed ottenere una struttura variabile ad ogni generazione.

2.1.1 MODULI E CREAZIONE DELLA LORO STRUTTURA DI BASE



Imm.2: logica di base dei tre moduli.

Si considerano tre tipologie di strutture modulari, rappresentate a Imm.2. I nodi raffigurati con la lettera 'x' rappresentano i nodi regolatori, ovvero quelli che hanno maggior potere nel modulo e si dice che regolano gli altri nodi. I nodi rappresentati con 'y' sono i nodi regolati, hanno quindi ruolo minore nella struttura e sottostanno ai regolatori. Le frecce continue rappresentano connessioni dirette e la loro direzione indica il verso della regolazione. Le frecce tratteggiate invece sono connessioni possibili che verranno aggiunte in seguito e analizzate al capitolo 2.1.3.

Il modulo uno (Imm.2) rappresenta una regolazione a feedback; si ha un nodo x di partenza che connette il primo y, la connessione prosegue in questo senso per tutti i nodi regolati fino ad arrivare a y_n, che è connesso al nodo regolatore. Il numero di nodi possibili in questa struttura è compreso tra uno e cinque.

Il modulo due (Imm.2) ha un nodo x connesso a tutti i nodi regolati y; questa struttura potrà anch'essa avere al massimo cinque nodi e come minimo due.

Il terzo modulo (Imm.2) utilizza un numero m di nodi x e una quantità n di nodi y, connettendoli in modo tale che ogni nodo x regoli almeno un nodo y e ogni nodo y sia regolato da almeno un nodo x.

Questo modulo ha dieci come numero massimo di nodi e due come minimo, in modo tale da descrivere più appropriatamente la complessità di queste strutture.
Si osserva ora il codice che porta alla generazione di queste tre strutture di base, andando a descrivere le tre funzioni che creano i moduli, salvate nella libreria 'modules'. Durante la descrizione del codice, una volta definito il significato di una variabile, si utilizzerà il suo nome per parlare di quel concetto, senza riscrivere cosa rappresenta per esteso. Questa idea si utilizzerà per semplificare le descrizioni ed evitare di appesantirle con ripetizioni inutili. Le variabili che assumono lo stesso significato in più funzioni, una volta introdotte subiranno lo stesso trattamento. Le descrizioni si riferiranno alle immagini contenenti codice e alle linee presenti in esse. La scrittura 'l.' seguita da uno o più numeri indica le diciture 'linea' o 'linee' a seconda del contesto.

```
5   def createmod1(Cref, nodesavailable):
6       if nodesavailable > 5:
7           N = 5
8       else:
9           N = nodesavailable
10      m = round(random.uniform(1,N))
11      connmat = []
12      x = [0]
13      y = []
14      if m != 1:
15          y = list(range(1,m))
16          for i in range(m):
17              connmat.append([False]*m)
18              if i == m-1:
19                  connmat[i][0] = True
20              else:
21                  connmat[i][i+1] = True
22      else:
23          connmat = [[False]]
24      connmat = updatemodule(connmat, x, y, 2, Cref)
25      return(connmat, x, y)
```

Imm.3: codice della funzione 'createmod1'

La prima funzione della libreria 'modules' è 'createmod1' (Imm.3) e il suo scopo è quello di generare la struttura di un modulo di tipologia uno.
Richiede in input il valore del coefficiente di clustering di riferimento 'Cref', che viene dato in input dall'utente attraverso il programma main, e il numero di nodi attualmente disponibili ad essere connessi al momento corrente 'nodesavailable'. A l.6-9 viene impostato il numero massimo 'N' di nodi che potranno formare il modulo. Di base come detto in precedenza si ha N=5, ma se 'nodesavailable' è minore di cinque viene assegnato N=nodesavailable. A l.10 viene definito il numero di nodi del modulo 'm' attraverso una generazione casuale di un valore tra uno e N ottenuto con la funzione 'uniform' della libreria 'random', che viene poi arrotondato all'intero più vicino dalla funzione 'round'. Alle linee 11-13 vengono inizializzate tre variabili:

- 'connmat' conterrà la matrice di connessioni del modulo sottoforma di valori bool (se è presente una connessione tra il nodo a e il nodo b si avrà connmat[a][b]=True, altrimenti il valore in tale posizione è False);

- ‘x’ conterrà gli indici dei nodi regolatori nella matrice di connessione. Viene posto x=[0] perché un modulo di tipologia uno avrà sempre e solo un nodo regolatore che definiamo essere il primo;
- ‘y’ conterrà gli indici dei nodi regolati.

A l.14-23 viene generata la struttura di base del modulo. Se il modulo è formato da solo un nodo (l.14) la funzione passa direttamente a l.23 e viene assegnata a ‘connmat’ la matrice di connessione di un singolo nodo (un unico valore False). In tutti gli altri casi invece viene riempito il vettore ‘y’ elencando i valori interi da uno ad m-1 con la funzione ‘range’ (l.15). Successivamente viene riempita ‘connmat’ scorrendo sui valori da zero a m-1 (quindi sugli indici dei nodi del modulo). Ad ogni iterazione viene aggiunta una riga con m valori False (nessuna connessione) che rappresenta le connessioni del nodo i-esimo considerato nel ciclo. Poi, per l’ultimo nodo del modulo (quindi all’ultima iterazione del ciclo) viene assegnata una connessione sulla colonna zero, che rappresenta la connessione tra il nodo y_n e il nodo regolatore. Per tutti gli altri nodi invece viene assegnata una connessione con il nodo che ha indice esattamente successivo, creando così la struttura richiesta del modulo uno. La Tab.2 rappresenta il caso in cui m=3.

	0	1	2
0	False	True	False
1	False	False	True
2	True	False	False

Tab.2: esempio di matrice di connessione per un modulo di tipologia tre, caso con tre nodi

A l.24 viene chiamata la funzione ‘updatemodule’ che aggiorna il modulo con le connessioni tratteggiate aggiuntive viste su Imm.2. La funzione ‘updatemodule’ verrà analizzata al capitolo 2.1.3. ‘createmod1’ restituisce come output ‘connmat’, ‘x’ e ‘y’.

```
28  def createmod2(Cref, nodesavailable):
29      if nodesavailable > 5:
30          N = 5
31      else:
32          N = nodesavailable
33      m = round(random.uniform(2,N))
34      connmat = []
35      x = [0]
36      y = list(range(1,m))
37      for i in range(m):
38          connmat.append([False]*m)
39          if i == 0:
40              for j in range(m):
41                  if j != 0:
42                      connmat[i][j] = True
43      connmat = updatemodule(connmat, x, y, 1, Cref)
44      return(connmat, x, y)
```

Imm.4: codice della funzione ‘createmod2’

La seconda funzione di ‘modules’ è ‘createmod2’ (Imm.4). Ha come obiettivo quello di generare un modulo di tipologia due. Si analizzano solo l.39-42, in quanto l’impostazione di questa funzione è molto simile a quella di ‘createmod1’.

L’unica porzione di codice diversa è ovviamente quella in cui vengono create le connessioni nel modulo. In questa struttura vengono assegnate delle connessioni solo alla prima iterazione del ciclo in cui si scorre tra i nodi del modulo (quindi solo quando si considera il nodo regolatore). Passata la condizione a l.39, vengono assegnati valori True in ‘connmat’ in tutte le posizioni ad esclusione della prima, in quanto il modulo prevede che il nodo regolatore sia connesso a tutti gli altri ma non a se stesso.

```
46 def createmod3(Cref, nodesavailable, gamma):
47     if nodesavailable > 10:
48         N = 10
49     else:
50         N = nodesavailable
51     m = round(random.uniform(2,N))
52     n1 = round(random.uniform(1,m-1))
53     x = list(range(n1))
54     n2 = m-n1
55     y = list(range(n1,m))
56     connmat = []
57     for i in range(m):
58         connmat.append([False]*m)
59     notsampled = list(y)
60     optimaldistry = distrD(gamma,n2)
61     for i in range(n1):
62         numconn = len(random.choices(range(1, n2+1), optimaldistry))
63         conn = random.sample(y, numconn)
64         for j in conn:
65             connmat[i][j] = True
66             if j in notsampled:
67                 notsampled.remove(j)
68     if len(notsampled) != 0:
69         optimaldistrx = distrD(gamma,n1)
70         for i in notsampled:
71             numconn = len(random.choices(range(1, n1+1), optimaldistrx))
72             conn = random.sample(x, numconn)
73             for j in conn:
74                 connmat[j][i] = True
75     connmat = updatemodule(connmat, x, y, 1, Cref)
76     return(connmat, x, y)
```

Imm.5: codice della funzione ‘createmod3’

L’ultima funzione di ‘modules’ è ‘createmod3’ (Imm.5) e genera un modulo di tipologia tre.

La creazione del terzo modulo è più complessa e richiede più passaggi degli altri due. Inizialmente la funzione richiede come input aggiuntivo il valore ‘gamma’, che rappresenta il parametro della distribuzione power-law, dato in input dall’utente attraverso il programma main. Una volta definito ‘N’ (in questo modulo il massimo standard sarà dieci come detto in precedenza) e ‘m’, a l.52-55 vengono generati il numero di nodi regolatori ‘n1’ e di nodi regolati ‘n2’ e i rispettivi vettori. ‘n1’ viene generato in modo simile a ‘m’, mentre ‘n2’ sarà dato dalla differenza tra m e n1, in modo tale che il numero totale di nodi del modulo sia n1+n2=m. A l.59 viene inizializzato un vettore ‘notsampled’ che contiene tutti i nodi y e rappresenta i nodi regolati nel modulo che non sono connessi

a nessun nodo x. Questo vettore serve perché successivamente verranno assegnate delle connessioni casuali ai nodi x e nel caso che uno o più nodi y non abbiano nessuna connessione, per soddisfare le richieste della formazione del modulo (ogni nodo y deve essere regolato da almeno un nodo x) si andranno a generare delle connessioni anche per quei nodi. A l.60 si genera un vettore 'optimaldistry' che contiene la distribuzione richiesta del grado di connessione dei nodi del network, utilizzando la funzione 'distrD' della libreria 'distributions' che sarà analizzata nel capitolo 2.2.1. A l.61-67 la funzione scorre sui nodi x e collega ognuno di essi a dei nodi y. Per farlo per ogni nodo viene generato un numero 'numconn' di connessioni casuale scelto con probabilità proporzionale a 'optimaldistry', utilizzando la funzione 'choices' della libreria 'random' (l.62). Poi vengono scelti numconn nodi da y attraverso la funzione 'sample' della libreria 'random' e salvati nel vettore 'conn' (l.63). Per ogni nodo in 'conn', viene creata una connessione tra il nodo regolatore e il nodo di 'conn' (l.65). A l.67 viene rimosso da 'notsampled' il nodo di 'conn' a cui è stata assegnata una connessione, ovviamente solo se si trova nel vettore (condizione a l.66) e quindi non ha già ricevuto una connessione in un'iterazione passata del ciclo. Una volta create delle connessioni per tutti i nodi x, la funzione aggiunge connessioni ai nodi di 'notsampled' se il vettore non è vuoto. Per ogni nodo di 'notsampled' vengono create delle connessioni con modalità identica a l.60-65, solo che in questo caso viene utilizzato 'notsampled' al posto di 'x', 'n1' al posto di 'n2' e non vengono rimossi da 'notsampled' i nodi y che ricevono delle connessioni in quanto non necessario.

2.1.2 FUNZIONI AUSILIARIE ALLA GENERAZIONE DELLA STRUTTURA FINALE DEL MODULO

A questo punto, dopo aver descritto il codice che permette la formazione delle strutture di base dei tre moduli, si analizza come il programma permette di aggiungere le connessioni aggiuntive tratteggiate ad Imm.2. Queste connessioni rappresentano ulteriori meccanismi di regolazione all'interno del modulo, attuati dal nodo x nei moduli di tipologia uno e tra diversi nodi regolati y in tutti gli altri. Queste due tipologie di connessioni aggiuntive sono descritte nel codice dal parametro 'typeconn'. Per i moduli due e tre, in cui le connessioni aggiuntive sono del tipo y->y, si ha typeconn=1. Per il modulo uno invece, in cui queste connessioni sono x->y, si ha typeconn=2. La funzione che permette di aggiungere alla struttura del modulo queste connessioni è 'updatemodule', chiamata in conclusione delle tre funzioni viste al capitolo 2.1.1. Tuttavia, prima di poter descrivere il suo funzionamento, è necessario capire come lavorano delle altre funzioni ausiliari essenziali per l'esecuzione di 'updatemodule'. Queste funzioni sono:

- 'obtainneighbors';
- 'pcneighbors' ;

- la libreria ‘cmcalc’ che contiene cinque funzioni, di cui tre vengono utilizzate da ‘updatemodule’.

La prima, ‘obtainneighbors’ (Imm.6), serve ad ottenere i vicini (neighbors) di un nodo ‘node’ presente all’interno di una struttura con matrice di connessione ‘mat’.

```
1  def obtainneighbors(mat, node):
2      neighbors = []
3      m = len(mat)
4      for i in range(m):
5          if mat[node][i]:
6              neighbors.append(i)
7          if mat[i][node] and i not in neighbors:
8              neighbors.append(i)
9      return neighbors
```

Imm.6: codice della funzione ‘obtainneighbors’

La funzione richiede in input ‘mat’ e l’indice del nodo ‘node’ nella struttura. A l.2 viene inizializzato il vettore ‘neighbors’ che conterrà i vicini di ‘node’. A l.4-8 viene effettuato uno scorrimento sui nodi della struttura. Se in posizione node[i] di ‘mat’ si trova un valore True (l.5-6), allora viene aggiunto a ‘neighbors’ il nodo i, perché tale risultato significa che ‘node’ è connesso (e regola) i. Successivamente a l.7 viene osservato il punto inverso della matrice e quindi viene controllato se i è connesso a node (e quindi node viene regolato da i). Se i non si trova già in ‘neighbors’ viene aggiunto. In conclusione, la funzione restituisce come output il vettore ‘neighbors’.

La seconda funzione ausiliaria è ‘pcneighbors’ (Imm.7). Il suo scopo è quello di calcolare quante sono le connessioni possibili tra i vicini di ‘node’ appartenente a ‘mat’.

Prima di analizzare la funzione però, è necessario definire alcune regole per quanto riguarda le connessioni tra nodi nei sistemi biologici considerati:

(a) non esiste l’autoregolazione, quindi non è possibile avere un nodo connesso a se stesso.

(b) Se esiste già una connessione tra un nodo a e un nodo b non si può generare una seconda connessione tra i due, in nessuna direzione (l’unica eccezione si ha per m=2 nei moduli di tipologia uno, in cui i due nodi si regolano a vicenda).

(c) Un nodo regolato y non può mai connettersi ad un nodo regolatore x, quindi non può esistere una connessione del tipo y->x (l’unica eccezione si ha per l’ultimo nodo del modulo uno).

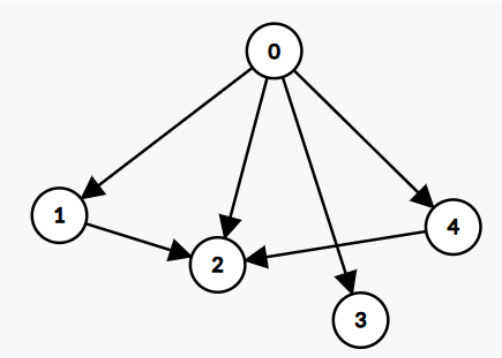
Definite queste regole si analizza ‘pcneighbors’.

```
3   def pcneighbors(mat,regulators,node,typeconn):
4       neighbors = obtainneighbors(mat,node)
5       x = []
6       y = []
7       for i in neighbors:
8           if i in regulators:
9               x.append(i)
10          else:
11              y.append(i)
12      n = 0
13      if typeconn == 1:
14          if len(y) <= 1:
15              return(n)
16          y.sort()
17          m = len(y)
18          newmat = []
19          for i in range(m):
20              newmat.append([False]*m)
21          for i in range(m):
22              for j in range(m):
23                  newmat[i][j] = mat[y[i]][y[j]]
24          ntrue = 0
25          for i in range(m):
26              for j in range(m):
27                  if i == j:
28                      continue
29                  if newmat[i][j]:
30                      ntrue += 1
31                  else:
32                      n += 1
33          n += -ntrue
34          n = n/2
35      else:
36          if len(x) == 0 or len(y) == 0:
37              return(n)
38          y.sort
39          x.sort
40          mx = len(x)
41          my = len(y)
42          newmat = []
43          for i in range(mx):
44              newmat.append([False]*my)
45          for i in range(mx):
46              for j in range(my):
47                  newmat[i][j] = mat[x[i]][y[j]]
48                  if mat[y[j]][x[i]]:
49                      n += -1
50          for i in range(len(newmat)):
51              for j in range(len(newmat[0])):
52                  if newmat[i][j]:
53                      continue
54                  else:
55                      n += 1
56          if n < 0:
57              n = 0
58      return(n)
```

Imm.7: codice della funzione ‘pcneighbors’

La funzione richede come input ‘mat’, i nodi regolatori ‘regulators’, ‘node’ e ‘typeconn’. A l.4 sono ottenuti i vicini di ‘node’ attraverso ‘obtainneighbors’. Essi vengono salvati nel vettore ‘neighbors’. A l.7-11 la funzione scorre tutti i nodi in ‘neighbors’ e in base al loro ruolo (ottenibile verificando se il nodo considerato si trova o meno dentro a ‘regulators’ (l.8)) li divide nei vettori ‘x’ e ‘y’; poi procede con il calcolo delle connessioni possibili tra questi nodi. A l.13 viene controllato quale tipo di connessione si può instaurare nel modulo, se typeconn=1 il programma continua come segue. Innanzitutto, viene considerato il caso di un modulo con un singolo nodo y (l.15), in tale situazione

nessuna connessione aggiuntiva sarà possibile e quindi la funzione restituirà immediatamente il valore zero. Negli altri casi viene preso ‘y’, ordinato con la funzione ‘sort’ (l.16) e viene creata una matrice ausiliaria ‘newmat’ (l.19-23), che rappresenta la matrice di connessione tra i nodi y, gli unici che possono creare connessioni aggiuntive nella struttura. A l.25-32 viene effettuato uno scorrimento su tutti gli elementi della matrice e aggiornata la variabile ‘n’ (che rappresenta il numero di connessioni disponibili) per ogni valore False incontrato (esclusi gli elementi della matrice sulla diagonale, che per la regola (a) non possono mai rappresentare connessioni) e la variabile ‘ntrue’ per ogni valore True incontrato. Una volta concluso il ciclo saranno noti il numero di connessioni già presenti ‘ntrue’ e il numero di connessioni disponibili ‘n’. In realtà questo non è esatto, perché per (b) il numero di connessioni disponibili non è n.



Imm.8: esempio di struttura modulare

nodo	0	1	2	3	4
0		X	X	X	X
1			X		
2					
3					
4			X		

Tab.3: matrice di connessione del modulo ad Imm.8. Sono segnalati con ‘X’ i punti in cui è presente una connessione, colorato in verde i punti in cui è possibile aggiungere una connessione, in rosso dove non si può per (a), (b) e (c) e per ‘typeconn’, che in questo caso è due.

Preso come esempio il modulo ad Imm.8 e la rispettiva matrice di connessione a Tab.3, se si esegue ‘pcneighbors’, una volta concluso il ciclo di l.25, risulta n=10 e ntrue=2. Però per (b) non è possibile avere una connessione in posizione (2,1) e (2,4), ovvero non ci possono essere connessioni nelle posizioni a indici inversi delle posizioni conteggiate da ‘ntrue’. Quindi il numero effettivo di connessioni possibili sarà dato da n-ntrue=8 (l.33) ed esattamente, sempre per (b) da (n-ntrue)/2=4 (l.34). Questo perché, per esempio, generando una connessione in (3,4), immediatamente (4,3)

diventa non disponibile e quindi ogni volta che una connessione viene aggiunta un'altra posizione si blocca. Ne consegue che se gli spazi disponibili per creare connessioni sono otto, in realtà il numero di connessioni che possono essere aggiunte al modulo sono la metà, quattro. Proseguendo nella lettura del codice a Imm.7, nel caso in cui typeconn=2 (l.36-55), il processo è simile a quello utilizzato per typeconn=1. L'unica differenza è che per (c) non serve fare il controllo di 'ntrue' come nel caso precedente, in quanto non ci possono essere connessioni y->x che non permettono di generare per (b) una nuova connessione x->y. Quindi il valore di 'n' ottenuto è effettivamente il numero di connessioni disponibili. Si ha un solo caso specifico che non permette a questa strategia di funzionare, ovvero l'ultimo nodo di ogni modulo uno, che permette una connessione y->x. Per ovviare a questo problema, nel caso in cui durante la creazione della matrice ausiliaria si verifichi questa situazione (l.48-49), viene aggiunto il valore -1 a 'n', in modo tale che nel conteggio finale risulti un valore in meno. Poi, in conclusione alla funzione (l.56-57), se 'n' è negativo lo si rende uguale a zero. Questo passaggio è necessario perché nel caso in cui non ci sia nessuna connessione possibile nella struttura, il valore di 'n' risulterebbe negativo, il che non è possibile. La funzione si conclude dando come output 'n'.

Si passa ora ad analizzare la libreria 'cmcalc'. Al suo interno sono presenti tutti gli strumenti necessari per calcolare il coefficiente medio di clustering, essenziale per l'aggiornamento del modulo attraverso 'updatemodule' e per molti altri processi che si considereranno in seguito. Per ottenere il valore C è necessario calcolare due vettori:

- 'K', che contiene nodo per nodo il grado di connessione di ogni elemento della struttura considerata. In questo vettore non c'è distinzione tra connessioni in-degree e out-degree.
- 'q', che contiene nodo per nodo il numero di connessioni presenti tra i vicini di ogni elemento della struttura.

Una volta ottenuti questi valori è possibile calcolare il coefficiente di clustering per ogni nodo e di conseguenza il valore medio all'interno della struttura.

```
3  def calcK(mat):
4      m = len(mat)
5      if m == 1:
6          return([0])
7      else:
8          K = [0]*m
9          for i in range(m):
10             Ki = 0
11             for j in range(m):
12                 if mat[i][j]:
13                     Ki += 1
14                 if mat[j][i]:
15                     Ki += 1
16             K[i] = Ki
17         return(K)
```

Imm.9: codice della funzione 'calcK'

La prima funzione della libreria è ‘calcK’ (Imm.9). Essa serve ad ottenere il vettore ‘K’.

La funzione richiede come input solamente la matrice di connessione della struttura ‘mat’. Inizialmente viene analizzato il caso in cui il numero di nodi della struttura sia solo uno (l.5). In tale situazione la funzione restituisce come output un vettore con un singolo valore zero, il nodo ovviamente non avrà nessun tipo di connessione e quindi grado nullo. Nel resto dei casi viene utilizzato ‘K’ (l.8). Per riempire il vettore, la funzione scorre tutti i valori della matrice per colonna e per riga nello stesso ciclo. Per ogni coppia riga/colonna (i,j) e colonna/riga (j,i) viene inizializzata una una variabile ‘Ki’ (l.10) a zero, che rappresenta il grado di connessione del singolo nodo. Se scorrendo per riga si incontra un valore True (l.12) il nodo ha una connessione out-degree e quindi il valore ‘Ki’ viene aggiornato. Lo stesso avviene se scorrendo per colonna si incontra un valore True (l.14), però in questo caso si avrà una connessione in-degree. Una volta analizzate le due coppie (i,j) e (j,i) Ki rappresenterà il grado di connessione corretto del nodo i, tale valore verrà quindi salvato in posizione i del vettore K (l.16). Come output la funzione restituisce ‘K’.

```
19  def calcq(mat):
20      m = len(mat)
21      q = [0]*m
22      if m == 1:
23          return [0]
24      else:
25          for i in range(m):
26              neighbors = []
27              qi = 0
28              neighbors = obtainneighbors(mat, i)
29              for node1 in neighbors:
30                  for node2 in neighbors:
31                      if mat[node1][node2]:
32                          qi += 1
33              q[i] = qi
34          return(q)
```

Imm.10: codice della funzione ‘calcq’

La seconda funzione è ‘calcq’ (Imm.10), serve per ottenere il vettore ‘q’.

Per farlo chiede come input solamente ‘mat’. Come su ‘calcK’, se ‘mat’ ha un unico nodo il programma restituisce un vettore con un solo elemento zero (l.22-23), altrimenti procede come segue. Scorrendo tra tutti i nodi del modulo viene inizializzato un vettore vuoto ‘neighbors’ e un valore ‘qi’ che rappresenta il numero di connessioni tra vicini del nodo i-esimo (l.26-27). A l.28 ‘neighbors’ viene riempito per il nodo i-esimo utilizzando la funzione ‘obtainneighbors’. Successivamente, controllando tutti i punti di ‘mat’ in cui potrebbero esserci delle connessioni tra i nodi di ‘neighbors’ (l.29-32), se viene incontrata una connessione si aggiorna ‘qi’ (l.32). Una volta concluso il ciclo for si assegna alla posizione i-esima di ‘q’ il valore ‘qi’. La funzione restituisce come risultato ‘q’.

```
36  def calcCm(m, K, q):
37      Cm = 0
38      C = [0]*m
39      if m == 1:
40          return Cm
41      else:
42          for i in range(m):
43              if K[i] == 1 or K[i] == 0:
44                  C[i] = 0
45                  continue
46              C[i] = (2*q[i])/(K[i]*(K[i]-1))
47          Cm = sum(C)/m
48          return(Cm)
```

Imm.11: codice della funzione ‘calcCm’

La terza funzione di ‘cmcalc’, ‘calcCm’ (Imm.11), utilizzando due funzioni appena definite calcola il coefficiente di clustering ‘Cm’ medio.

Richiede come input il numero di nodi ‘m’ della struttura considerata e i vettori ‘K’ e ‘q’. A l.38 viene inizializzato il vettore ‘C’, che conterrà per ogni nodo il suo coefficiente di clustering. Se la struttura ha un solo nodo la funzione restituisce come risultato il valore zero (l.39-40), altrimenti per ogni nodo calcola il valore del coefficiente di clustering. Per evitare problemi con la divisione per zero, se il valore ‘K’ del nodo attuale è uguale a uno o nullo, viene assegnato in automatico come valore ‘C’ del nodo i-esimo zero e la funzione passa al nodo successivo (l.43-45). In tutti gli altri casi viene calcolato ‘C’ con Eq.2 (l.46). Concluso il calcolo di tutti gli elementi di ‘C’, ‘Cm’ è ottenuto facendo la media tra i valori di ‘C’ (l.47). La funzione restituisce come risultato ‘Cm’.

Le ultime due funzioni di ‘cmcalc’ non vengono utilizzate da ‘updatemodule’. Tuttavia, avendo scopi molto simili a ‘calcK’ sono contenute dentro a ‘cmcalc’ e per completezza verranno descritte ora.

```
50  def calcKid(mat):
51      m = len(mat)
52      if m == 1:
53          return([0])
54      else:
55          Kid = [0]*m
56          for i in range(m):
57              for j in range(m):
58                  if mat[j][i]:
59                      Kid[i] += 1
60      return(Kid)
```

```
62  def calcKod(mat):
63      m = len(mat)
64      if m == 1:
65          return([0])
66      else:
67          Kod = [0]*m
68          for i in range(m):
69              for j in range(m):
70                  if mat[i][j]:
71                      Kod[i] += 1
72      return(Kod)
```

Imm.12: codice delle funzioni ‘calcKid’ e ‘calcKod’

Le funzioni ‘calcKid’ e ‘calcKod’ (Imm.12) servono rispettivamente a calcolare il vettore ‘K’ per il solo grado di connessione in-degree della struttura data e il grado di connessione out-degree. La logica utilizzata e la struttura della funzione è la stessa di ‘calcK’, solo che anziché conteggiare entrambe le tipologie di connessioni nel calcolo, la prima funzione considera solo le connessioni in-degree (l.58-59) e la seconda solo le out-degree (l.70-71).

Ora che è stato definito il funzionamento di ‘calcK’, ‘calcq’ e ‘calcCm’ si può proseguire con il codice della funzione ‘updatemodule’, che a questo punto può essere analizzata in modo completo.

2.1.3 GENERAZIONE DELLA STRUTTURA FINALE DEI MODULI

La funzione ‘updatemodule’ ha il compito di aggiungere le connessioni tratteggiate nel modulo in base a ‘typeconn’. Questo viene fatto per modificare il coefficiente medio di clustering del modulo, perché si vuole rendere tale valore il più vicino possibile al coefficiente di riferimento ‘Cref’. Aggiungendo nuove connessioni il coefficiente di clustering aumenta e una volta raggiunto o superato ‘Cref’ si ferma. Per fare ciò, la funzione scorre nodo per nodo gli elementi del modulo e se possibile crea connessioni tra i vicini del nodo considerato.

```
6   def updatemodule(mod, x, y, typeconn, Cref):
7       m = len(mod)
8       K = calcK(mod)
9       q = calcq(mod)
10      Cm = calcCm(m, K, q)
11      nodes = x + y
12      nodestoremove = []
13      for i in nodes:
14          pc = pcneighbors(mod,x,i,typeconn)
15          if pc == 0:
16              nodestoremove.append(i)
17      for i in nodestoremove:
18          nodes.remove(i)
19      nodestoremove = []
20      while Cm < Cref and len(nodes) != 0:
21          random.shuffle(nodes)
22          for i in nodes:
23              qim = 0.5*Cref*K[i]*(K[i]-1)
24              qis = round(random.gauss(qim,1))
25              if qis > q[i]:
26                  newlinks = qis - q[i]
27                  pc = pcneighbors(mod,x,i,typeconn)
28                  if newlinks > pc:
29                      if pc == 0:
30                          nodestoremove.append(i)
31                          continue
32                      else:
33                          newlinks = pc
34                  if typeconn == 1:
35                      while newlinks != 0:
36                          n1 = random.choice(y)
37                          n2 = random.choice(y)
38                          if n1 == n2 or mod[n1][n2] or mod[n2][n1]:
39                              continue
40                          else:
41                              mod[n1][n2] = True
42                              newlinks += -1
43                  else:
44                      while newlinks != 0:
45                          n1 = random.choice(x)
46                          n2 = random.choice(y)
47                          if mod[n1][n2] or mod[n2][n1]:
48                              continue
49                          else:
50                              mod[n1][n2] = True
51                              newlinks += -1
52              K = calcK(mod)
53              q = calcq(mod)
54          for j in nodestoremove:
55              nodes.remove(j)
56          nodestoremove = []
57          Cm = calcCm(m, K, q)
58      return(mod)
```

Imm.13: codice delle funzione ‘updatemodule’

La funzione (Imm.13) richiede in input la matrice di connessione del modulo ‘mod’, i vettori ‘x’ e ‘y’ e i parametri ‘typeconn’ e ‘Cref’. A l.8-10 vengono calcolati attraverso le funzioni ‘calcK’, ‘calcq’ e ‘calcCm’ questi tre valori del modulo. A l.12-19 la funziona genera un vettore ‘nodestoremove’ che conterrà tutti i nodi che non possono creare nuove connessioni tra i propri vicini e che quindi è inutile considerare. A l.13-16 viene controllato quali sono questi nodi calcolando il numero di connessioni possibili tra vicini ‘pc’ attraverso la funzione ‘pcneighbors’ (l.14). I nodi che hanno pc=0 vengono aggiunti a ‘nodestoremove’ (l.15-16). Poi a l.17-18 vengono rimossi da ‘nodes’ e a l.19 viene svuotato ‘nodestoremove’. Quest’ultima operazione viene fatta perché aggiungendo connessioni nel modulo è possibile che alcuni nodi non possano più generarne di nuove, quindi ‘updatemodule’ continuerà ad utilizzare questo vettore ad ogni iterazione per togliere da ‘nodes’ tutti i nodi che non serve più controllare. La funzione è basata su un ciclo while (l.20) che fino a che ‘Cm’ è minore di ‘Cref’ cerca di aggiungere connessioni, a meno che ‘nodes’ sia vuoto e quindi le connessioni possibili aggiuntive si sono esaurite. Dentro al ciclo, attraverso la funzione ‘shuffle’ della libreria ‘random’, vengono ordinati casualmente i nodi di ‘nodes’ (l.21). Questo viene fatto per evitare di scorrere i nodi sempre nello stesso ordine, il che sarebbe un problema perché il programma cercherebbe ogni volta di aggiungere connessioni ai primi nodi e di conseguenza non avrebbe un processo di addizione casuale. Una volta definito il vettore ‘nodes’, la funzione inizia a scorrere tra i suoi elementi e ad ogni iterazione calcola il valore ‘qim’ (l.23) che rappresenta la quantità esatta di connessioni che dovrebbero avere i vicini del nodo i-esimo per portare il coefficiente di clustering del nodo i al valore Cref. Siccome l’obiettivo del progetto è generare un network casuale, a l.24 viene determinato un valore ‘qis’, ottenuto dato da una distribuzione gaussiana con media qim e deviazione standard uguale a uno (attraverso la funzione ‘gauss’ della libreria ‘random’), da cui viene scelto casualmente un valore e arrotondato all’intero più vicino. A questo punto, se qis è maggiore del numero di connessioni attualmente presenti tra i vicini di i (l.25), la funzione procede aggiungendo una quantità ‘newlinks’ di connessioni (l.26), date dalla differenza tra qis e q[i]. Prima di aggiungere le connessioni è necessario eseguire dei controlli, la funzione deve verificare che il numero di connessioni possibili tra i vicini non sia minore di newlinks. Quindi a l.19 viene calcolato ‘pc’ per il nodo i-esimo e se ‘newlinks’ è maggiore di ‘pc’ potranno verificarsi due casi:

- ‘pc’ è uguale a zero: in questo caso viene aggiunto il nodo i a ‘nodestoremove’ (l.29-30), per poi poterlo rimuovere da ‘nodes’ una volta concluso lo scorrimento sui nodi. Subito dopo la funzione passa al controllo del nodo successivo (l.31).
- ‘pc’ è non nullo: in questo caso ‘newlinks’ viene impostato uguale a ‘pc’ (l.33) e così verranno aggiunte il numero massimo di connessioni possibili.

Nel caso in cui ‘newlinks’ sia minore di ‘pc’ invece non c’è nessun controllo da eseguire.

Una volta stabilito ‘newlinks’ in modo definitivo, si procede con l’aggiunta delle connessioni. In base a ‘typeconn’ viene seguito un tipo di inserimento (l.35-42) o l’altro (l.44-51), ma il concetto è lo stesso, quindi verrà analizzato solo il primo, quello per connessioni y->y. La funzione, fino a che il valore di ‘newlinks’ non risulta nullo, continua a cercare di aggiungere delle connessioni tra i vicini del nodo i. Vengono selezionati due nodi a caso ‘n1’ e ‘n2’ tra i nodi ‘y’ (l.36-37) e poi viene eseguito un controllo per verificare se la connessione tra quei due nodi può essere fatta. Nei casi in cui n1 e n2 siano lo stesso nodo oppure la connessione tra n1 ed n2 sia già presente (in uno dei due sensi), si ritorna alla fase di scelta dei due nodi (l.38-39); altrimenti si crea la connessione tra n1 ed n2 (l.41) e si riduce di uno il valore di ‘newlinks’. L’unica differenza in questo processo per le connessioni x->y è che la condizione di controllo n1==n2 non è necessaria, in quanto scegliendo un nodo ‘n1’ da ‘x’ (l.45) e un nodo ‘n2’ da ‘y’ (l.46) non è possibile che i due siano lo stesso nodo. Una volta conclusa l’assegnazione di nuove connessioni al nodo i-esimo, vengono calcolati nuovamente i valori ‘K’ e ‘q’ (l.52-53), perché avendo aggiunto delle connessioni essi sono cambiati e per calcolare qim correttamente per il nodo successivo è necessario aggiornarli. Successivamente, concluse le iterazioni su tutti i nodi di ‘nodes’, vengono rimossi da questa lista i valori che sono stati aggiunti a ‘nodestoremove’, che poi viene svuotato (l.54-56). Viene calcolato Cm (l.57) e la funzione torna a l.20 per controllare se la condizione del while iniziale è ancora vera e quindi se è necessario aggiungere ulteriori connessioni. In conclusione, la funzione restituisce come risultato la matrice di connessione del modulo con tutte le connessioni aggiuntive.

Utilizzando ‘updatemodule’, le tre funzioni di ‘modules’ possono completare la generazione dei moduli inserendo le connessioni aggiuntive. Si passa ora alla seconda fase dell’esecuzione del programma, il calcolo dello score.

2.2 CALCOLO DELLO SCORE MODULARE E SELEZIONE DI UN MODULO DA UTILIZZARE

Ad ogni modulo che verrà generato durante l'esecuzione viene assegnato un punteggio (score). Questo valore indica quanto benefica è la presenza del modulo all'interno del network perché i parametri del grafo si avvicinino a quelli richiesti. Più è alto lo score, migliore sarà l'effetto del modulo sul network, più è basso più l'effetto sarà negativo. In questa fase dell'esecuzione, viene calcolato lo score per il modulo generato, attraverso la funzione 'score'. Il punteggio è ottenuto utilizzando alcune distribuzioni, si passa ora alla loro definizione prima dell'analisi di 'score'.

2.2.1 CALCOLO DELLO SCORE

Come definito al capitolo 1.1, il grafo generato deve avere distribuzione del grado di connessione dei nodi proporzionale alla power-law descritta da Eq.1. Per calcolare uno score che spieghi quanto un modulo è adatto a migliorare questo parametro si definiscono tre distribuzioni, che permetteranno di calcolare un punteggio chiamato 'degree score'. La prima distribuzione (già incontrata nella funzione 'createmod3' ad Imm.5) rappresenta la distribuzione desiderata D del grafo, che è ottenuta proporzionalmente alla power-law. È data da:

$$D = [d_1, d_2, \dots, d_N,] \; con \; d_K \propto \frac{1}{K^\gamma}$$

Eq.3: distribuzione desiderata del grafo

In cui K è il valore del grado connessione e ogni elemento d_K la proporzione con cui esso deve essere presente tra i nodi del grafo. La seconda distribuzione rappresenta la distribuzione attuale P di una struttura interconnessa e quindi il modo in cui il grado di connessione è distribuito tra i nodi. È data da:

$$P = [p_1, p_2, \dots, p_N,]$$

Eq.4: distribuzione attuale di una struttura

Infine, preso un grafo G, l'insieme V dei nodi di G che devono essere connessi tra loro (è il vettore V definito al capitolo 1.3) e un modulo M con m nodi, la terza distribuzione Fij rappresenta la distribuzione che G avrà in futuro una volta aggiunto M. Nello specifico considera il caso in cui il nodo i di V ricopre il ruolo del nodo j in M. Quindi esattamente indica la distribuzione che avrà il grafo sotto queste condizioni, con i avente ruolo j e connesso agli altri m-1 nodi del modulo. La distribuzione, per ogni coppia (i,j), è data da:

$$F_{ij} = [f_{1ij}, f_{2ij}, \dots, f_{Nij},]$$

Eq.5: distribuzione futura di una struttura per la coppia (i,j)

Una volta ottenute queste distribuzioni, preso un modulo M, per ogni coppia (i,j) è possibile calcolare uno score Sij che rappresenta quanto il modulo M per la coppia scelta (i,j) migliori la distribuzione attuale P del grafo avvicinandosi alla desiderata D. Calcolando il punteggio per ogni coppia (i,j), si ottiene lo score S, il 'degree score'. Esso è dato dalla sommatoria dei punteggi Sij e dice complessivamente quanto il modulo è adatto a migliorare i parametri del grafo.

$$S = \sum_{i=1}^{len(V)} \sum_{j=1}^{m} S_{ij}$$

Eq.6: calcolo del degree score

Per ogni coppia (i,j) il punteggio Sijk è definito per ogni grado di connessione K da Eq.7. Poi Sij è dato dalla sommatoria dei valori di Sijk.

$$S_{ijk} = sign\big(|d_k - p_k| - |d_k - f_{ijk}|\big) \cdot \frac{|d_k - p_k|}{d_k}$$

Eq.7: calcolo del degree score per il grado di connessione k nella coppia (i,j)

In Eq.7, il valore $sign\big(|d_k - p_k| - |d_k - f_{ijk}|\big)$ determina se Sijk ha valore positivo, negativo o nullo attraverso la funzione segno. Può avere tre valori:

- se il risultato è +1 significa che per il grado di connessione K la distribuzione futura f_{ijk} ha un valore più vicino alla distribuzione desiderata d_k rispetto a quella attuale. Quindi il nodo i con ruolo j diminuisce la distanza tra la frequenza ideale e attuale di occorrenza di nodi con grado k nel network e migliora le caratteristiche del grafo;
- se è 0, significa che la distribuzione attuale e futura sono le stesse e quindi la coppia (i,j) non apporta né miglioramenti né peggioramenti al network;
- se è -1 vuol dire che la distribuzione attuale ha valore più vicino a quella richiesta rispetto a quella attuale, quindi la coppia (i,j) porta ad un peggioramento delle caratteristiche del grafo.

Il valore $\frac{|d_k - p_k|}{d_k}$ a Eq.7 rappresenta il valore assoluto di Sijk, più è alta la distanza tra distribuzione attuale e desiderata, più questo valore sarà alto. Il termine d_k a denominatore si utilizza per bilanciare l'influenza della differenza a numeratore per valori di k bassi.

Una volta definito il degree score, si procede analizzando il codice che calcola le distribuzioni D, P e F e successivamente la funzione 'score' che ottiene il punteggio modulare.
Le tre distribuzioni sono ottenute da delle funzioni presenti nella libreria 'distributions', segue l'analisi della prima di esse, 'distrD' (Imm.14).

```
3   def distrD(gamma, N):
4       w = [0]*N
5       D = [0]*N
6       for i in range(N):
7           w[i] = 1/(pow(i+1,gamma))
8       S = sum(w)
9       for i in range(N):
10          D[i] = w[i]/S
11      return(D)
```

Imm.14: codice delle funzione ‘distrD’

Questa funzione serve per calcolare la distribuzione D.

Richiede in input il parametro ‘gamma’ della power-law e il massimo grado di connessione ‘N’ presente nella struttura considerata. Vengono inizializzati e riempiti il vettore ‘w’ dei pesi (contiene i valori della power law) e ‘D’ della distribuzione. ‘D’ è ottenuto normalizzando ‘w’ (l.8-10), perché la distribuzione del grado di connessione deve essere proporzionale ai valori della power-law, non seguirla esattamente. La funzione restituisce ‘D’ come output.

```
13  def distrP(mat, od_id):
14      if od_id == 1:
15          conn = calcKod(mat)
16      else:
17          conn = calcKid(mat)
18      maxconn = max(conn)
19      if maxconn == 0:
20          return([float(0)])
21      else:
22          P = [0]*maxconn
23      for i in range(len(conn)):
24          if conn[i] == 0:
25              continue
26          else:
27              P[conn[i]-1] += 1
28      S = len(conn)
29      for i in range(maxconn):
30          P[i] = P[i]/S
31      return(P)
```

Imm.15: codice delle funzione ‘distrP’

La seconda funzione, ‘distrP’ (Imm.15), calcola la distribuzione P.

Chiede in input la matrice di connessione ‘mat’ della struttura e il parametro ‘od_id’. Attraverso quest’ultimo, se od_id=1 la funzione otterrà la distribuzione P del grado di connessione out-degree, se invece od_id=2 lo farà per la in-degree. Dunque, in base a questo parametro, a l.14-17 viene calcolato un vettore contenente il grado di connessione ‘conn’ per ogni nodo (vettore K in o out-degree), utilizzando la funzione ‘calcKod’ (l.15) o ‘calcKid’ (l.17) della libreria ‘cmcalc’. Successivamente a l.18 la funzione ottiene il grado massimo di connessione ‘maxconn’ tra i nodi della struttura. Se ‘maxconn’ è zero il programma restituisce immediatamente il valore [0] (l.20), la struttura non ha connessioni e quindi la distribuzione è nulla per ogni grado di connessione. In questo caso è sufficiente annotare in ‘P’ che la distribuzione per K=1 è zero, non serve inserire valori nulli

per i successivi K. Negli altri casi, viene inizializzato il vettore ‘P’ con lunghezza ‘maxconn’ (l.22). Successivamente la funzione scorre i valori di ‘conn’ e per ogni elemento incontrato aggiorna ‘P’ in modo tale da avere in ogni posizione il numero di nodi con grado di connessione K+1 (l.27). ‘P’ è indicizzata in base a K+1 perché l’indicizzazione in python parte da 0 e qualsiasi nodo con grado di connessione zero non viene considerato (l.24-25). A l.28-30 ‘P’ viene normalizzata in base al numero di nodi della struttura, ottenendo così la distribuzione. La funzione restituisce in output ‘P’.

```
33  def distrFm(G, V, M, od_id):
34      m = len(M)
35      N = len(G)
36      if m == 1:
37          F = [[distrP(G, od_id)]*m]*len(V)
38          return(F)
39      if od_id == 1:
40          connG = calcKod(G)
41          connM = calcKod(M)
42      else:
43          connG = calcKid(G)
44          connM = calcKid(M)
45      maxconn = max(connM) + max(connG)
46      nPG = [0]*maxconn
47      for i in range(len(connG)):
48          if connG[i] == 0:
49              continue
50          else:
51              nPG[connG[i]-1] += 1
52      nPM = [0]*maxconn
53      for i in range(len(connM)):
54          if connM[i] == 0:
55              continue
56          else:
57              nPM[connM[i]-1] += 1
58      Fi_std = [0]*maxconn
59      for i in range(maxconn):
60          Fi_std[i] = nPM[i] + nPG[i]
61      for i in range(maxconn):
62          Fi_std[i] = Fi_std[i]/N
63      Fi_std = [Fi_std]*m
64      F = []
65      for i in V:
66          cG = connG[i]
67          Fi = []
68          if cG == 0:
69              Fi = Fi_std
70              F.append(Fi)
71              continue
72          for j in range(m):
73              nGnew = list(nPG)
74              cM = connM[j]
75              cGnew = cM + cG
76              for t in range(m):
77                  if t == j or connM[t] == 0:
78                      continue
79                  else:
80                      nGnew[connM[t]-1] += 1
81              if cGnew != cG:
82                  nGnew[cG-1] += -1
83                  nGnew[cGnew-1] += 1
84              for r in range(maxconn):
85                  nGnew[r] = nGnew[r]/N
86              Fi.append(nGnew)
87          F.append(Fi)
88      return(F)
```

Imm.16: codice delle funzione ‘distrFm’

L'ultima funzione di 'distributions', 'distrFm' (Imm.16), calcola le distribuzioni Fij per ogni coppia (i,j) possibile e le salva tutte in una lista di liste 'F'.
'distrFm' richiede in input la matrice di connessione 'G' del grafo e quella del modulo 'M', il vettore 'V' e il parametro 'od_id'. Inizialmente viene preso in considerazione il caso di modulo con un solo nodo (m=1). In questa situazione il modulo non andrà mai, per nessuna coppia (i,j) a modificare la distribuzione del grado di connessione del grafo, in quanto non può aggiungere connessioni ad esso. Di conseguenza, per qualsiasi coppia (i,j), Fij sarà sempre la stessa e di preciso è la distribuzione attuale P del grafo. Quindi il programma assegna a 'F', per ogni coppia (i,j), la distribuzione ottenuta utilizzando 'distrP' su 'G'(l.37) e poi restituisce immediatamente 'F' come output. Negli altri casi, in base a 'od_id', viene calcolato il vettore K per i nodi del grafo 'connG' e del modulo 'connM' (l.39-44).
Assegnando ai nodi i di V un ruolo j dentro a M, il grado di connessione di i potrebbe variare. Ad esempio, se i ha connG[i]=3 e gli si assegna il ruolo j con connM[j]=2, il nodo ottenuto avrà grado di connessione 3+2=5 e non più tre. Dunque, per capire come impostare le lunghezze dei vettori da utilizzare, a l.45 la funzione ottiene il numero massimo di connessioni 'maxconn' che potrà avere un nodo tra tutti i nodi generabili scorrendo tra le coppie (i,j). Questo viene fatto sommando il massimo valore di 'connG' con quello massimo di 'connM', perché ad un certo punto il nodo con grado massimo del grafo dovrà rivestire il ruolo del nodo con grado massimo nel modulo e quindi avrà grado dato dalla somma dei due massimi, che è il massimo valore ottenibile. A l.46-57, con le stesse modalità della funzione 'distrP' (Imm.15), la funzione calcola per il modulo e per il grafo il numero di nodi nelle strutture che hanno grado di connessione k, per ogni valore di k={1,…,maxconn}.
Subito dopo viene ottenuto 'Fi_std', che preso un nodo i di V rappresenta Fij per tutte le coppie (i,j) generate da i nel caso in cui sia un nodo senza connessioni. Se un nodo ha grado di connessione zero, per qualsiasi ruolo j ricopra, le uniche connessioni che verranno aggiunte al grafo sono quelle presenti su M. Dunque, per ogni coppia (i,j) per tale i, la distribuzione ottenuta è sempre la stessa ed è calcolabile sommando il numero di connessioni per ogni K già presenti nel grafo 'nPG' con quelle del modulo 'nPM' e poi normalizzando il risultato dividendo per il numero di nodi del grafo (l.58-63).
Subito dopo viene inizializzato 'F' vettore vuoto e la funzione inizia a scorrere sui nodi di V per calcolare Fij per ogni coppia. Viene ottenuto il numero di connessioni 'cG' del nodo considerato (l.66) e inizializzato 'Fi' vettore vuoto (l.67) che conterrà tutte le distribuzioni Fij ottenibili dal nodo i considerato. Se il nodo i non ha connessioni e quindi ricade nella categoria descritta a l.58-63, viene assegnato in automatico 'Fi_std' a 'Fi', aggiunto 'Fi' a 'F' e la funzione passa all'analisi del nodo

successivo (l.68-71). Se il nodo i non fa parte di questa tipologia viene eseguito lo scorrimento tra tutti i ruoli j che può avere e calcolata la relativa distribuzione nelle modalità descritte in seguito.
La funzione crea una nuova lista 'nGnew' (l.73), inizializzata identica a 'nPG', che rappresenterà il numero di nodi con grado di connessione K per ogni K se il nodo i riveste ruolo j. Poi viene ricavato il numero 'cM' di connessioni presenti sul nodo j del modulo (l.74) e calcolato il numero 'cGnew' di connessioni che il nodo i avrebbe se avesse ruolo j (l.75). Successivamente la funzione modifica 'nGnew' in base alle connessioni presenti sul modulo e al nuovo grado di connessione di i: deve aggiungere tutte le connessioni presenti sul modulo meno quelle di j che vanno trattate a parte. Vanno considerate separatamente perché possono modificare il grado di connessione di i. In tal caso è necessario sottrarre uno da 'nGnew' nella posizione corrispondente al vecchio grado di connessione di i e addizionare uno nella posizione di quello nuovo. Per aggiornare 'nGnew' con tutte le nuove connessioni correttamente, prima a l.76-80 vengono aggiunte le connessioni del modulo (l.80) ad esclusione di quelle di j. A l.81-83, se il numero di connessioni del nodo i che ha assunto ruolo j è cambiato, viene rimosso il vecchio grado di connessione da 'nGnew' e aggiunto quello nuovo. Poi 'nGnew' viene normalizzato e aggiunto a 'Fi' (l.84-86).
Una volta terminato il ciclo sui ruoli j e quindi riempito completamente 'Fi', esso viene aggiunto a 'F' (l.87) e il ciclo prosegue osservando il nodo i successivo. In conclusione, la funzione dà come output 'F'.
Si osserva che utilizzando vettori Fij di lunghezza 'maxconn', in tutti i casi ad esclusione di quello che presenta il nodo con grado massimo, i vettori delle distribuzioni Fij conterranno sempre dei valori zero inutili in coda. Tuttavia, per evitare di suddividere la funzione in troppe sezioni, vengono utilizzati vettori della stessa lunghezza, perché alla fine quando i dati ottenuti verranno utilizzati nella funzione 'score' questi zeri in più saranno irrilevanti e verranno ignorati.

Si passa ora ad analizzare 'score' (Imm.17), la funzione che calcola il degree score per i moduli generati. Come visto al capitolo 1.3, 'SimBioNeT' genera contemporaneamente un modulo per ognuna delle delle tre tipologie. Per questo motivo, la funzione 'score' permette di calcolare lo score anche per più di un modulo (non per forza tre). Infatti, può ricevere in input i dati di più moduli e ad ogni sua chiamata può calcolare più punteggi in una sola esecuzione.

```
5   def score(G,V,M,gamma,od_id):
6       nmodules = len(M)
7       P = distrP(G,od_id)
8       maxdgrP = len(P)
9       v = len(V)
10      S = [0]*nmodules
11      modsSij = ([[] for _ in range(nmodules)])
12      for r in range(nmodules):
13          F = distrFm(G, V, M[r],od_id)
14          m = len(M[r])
15          fij = len(F[0][0])
16          for i in range(v):
17              Si = [float(0) for _ in range(m)]
18              for j in range(m):
19                  Fij = F[i][j]
20                  maxdgrF = 1
21                  for k in range(fij-1,-1,-1):
22                      if Fij[k] != 0:
23                          maxdgrF = k+1
24                          break
25                  if maxdgrF > maxdgrP:
26                      maxdgr = maxdgrF
27                      P += [float(0)] * (maxdgr-len(P))
28                  elif maxdgrF == maxdgrP:
29                      maxdgr = maxdgrP
30                      pass
31                  else:
32                      maxdgr = maxdgrP
33                      Fij += [float(0)] * (maxdgr-len(F))
34                  D = distrD(gamma,maxdgr)
35                  for k in range(maxdgr):
36                      Si[j] += float(np.sign(abs(D[k]-P[k])-abs(D[k]-Fij[k]))*(abs(D[k]-P[k])/D[k]))
37                  S[r] += Si[j]
38              modsSij[r].append(Si)
39      PR = [0]*nmodules
40      if all(s <= 0 for s in S):
41          pass
42      else:
43          Stot = 0
44          for i in range(nmodules):
45              if S[i] > 0:
46                  Stot += S[i]
47          for i in range(nmodules):
48              if S[i] > 0:
49                  PR[i] = S[i]/Stot
50      return(PR, modsSij)
```

Imm.17: codice della funzione ‘score’

La funzione richiede in input la matrice di connessione del grafo ‘G’, ‘V’, la lista contenete le matrici di connessione dei vari moduli ‘M’, i parametri ‘gamma’ e ‘od_id’. A l.6-11 vengono definiti diversi dati, tra cui la distribuzione P del grafo (l.7), il grado di connessione massimo presente nel grafo ‘maxdgrP’ (l.8), viene inizializzato un vettore che conterrà i degree score dei vari moduli (l.10) e una lista ‘modsSij’ che conterrà tutti gli score Sij per i vari moduli (l.11). Ora per ogni modulo la funzione procede con il calcolo del degree score. Scorrendo i moduli, come primo passaggio, la funzione ottiene la distribuzione ‘F’ del grafo con il modulo attuale inserito (l.13) e il grado massimo di connessione ‘fij’ al suo interno (l.15). Ottenuti questi valori, la funzione scorre tutti i nodi i di V per calcolare gli score Sij di ogni coppia. Viene inizializzato un vettore ‘Si’ (l.17) che conterrà i punteggi di tutte le coppie generate dal nodo i, che poi servirà per aggiornare ‘modsSij’. Subito dopo inizia lo scorrimento tra i nodi j del modulo.

Per ogni coppia (i,j), la funzione ottiene i tre vettori delle distribuzioni e li imposta in modo tale che abbiano la stessa lunghezza. In questo modo la distribuzione D verrà calcolata con l'effettivo grado massimo di connessione e successivamente lo scorrimento che verrà effettuato per calcolare il singolo score non porterrà ad errori "out of bounds", perché tutti i vettori avranno lunghezza uguale al massimo grado di connessione. Viene ottenuta la distribuzione 'Fij' per la coppia selezionata (l.19) e inizializzato il grado massimo di connessione 'maxdgrF' di 'Fij' ad uno (l.20). Poi a l.21-24 la funzione scorre al contrario il vettore 'Fij' e continua a scorrere fino a che incontra un valore non nullo. In questo modo viene ottenuto il valore effettivo di 'maxdgrF'.

A questo punto vengono resi di uguale lunghezza i vettori 'P' e 'Fij'. In base a quale dei due è più lungo possono verificarsi tre casi:

- se 'maxdgrF' è maggiore di 'maxdgrP' (l.25-27) viene creata una variabile che rappresenta il grado massimo di connessione 'maxdgr' nel grafo con l'aggiunta del modulo per la coppia (i,j) (che sarà quindi uguale a 'maxdgrF') e vengono aggiunti valori zero al vettore 'P' in modo tale che abbia lunghezza 'maxdgr';
- se le due distribuzioni hanno già lo stesso grado massimo (l.28-30), si assegna 'maxdgr' uguale a una delle due e si continua;
- se 'maxdgrP' è maggiore di 'maxdgrF') a l.31-33, viene eseguito lo stesso procedimento di l.25-27, ma ovviamente con le due distribuzioni invertite.

Ottenendo i vettori in questa modalità, quasi sempre il vettore 'Fij' in realtà avrà una lunghezza maggiore degli altri, perché calcolando 'maxdgrF', il vettore 'Fij' non viene accorciato togliendo gli zeri superflui. Questo però non è un problema, perché successivamente, calcolando lo score a l.35-36, gli elementi nulli in più non verranno mai considerati.

A l.34 la funzione ottiene la distribuzione 'D' con grado massimo 'maxdgr' e poi per ogni grado di distribuzione, da uno a 'maxdgr', viene calcolato lo score Sijk attraverso Eq.7 e sommato al valore presente su 'Si' in posizione j (l.36), in modo tale che alla fine 'Si' conterrà i vari score Sij per il nodo i. Prima di passare al nodo j successivo viene aggiornato lo score complessivo del modulo nel vettore 'S' (l.37). Dopo aver calcolato lo score di tutte le coppie per il nodo i viene aggiunto a 'modsSij' il vettore 'Si' (l.38).

A questo punto è concluso il calcolo del degree score per i vari moduli, allora la funzione procede normalizzando il vettore 'S'. Questo passaggio viene eseguito perché i risultati di questa funzione verranno utilizzati per selezionare uno tra i moduli da aggiungere al network in base allo score. Dunque si utilizza un vettore normalizzato 'PR' in cui ogni elemento rappresenta la probabilità di scelta del modulo associato. La funzione inizializza tale vettore a l.39 e procede riempendolo.

Se tutti gli score in ‘S’ sono negativi o nulli, il vettore ‘PR’ è già a posto e la funzione può continuare la sua esecuzione senza modificarlo (l.40-41). Questo accade perché non volendo selezionare moduli che peggiorano le caratteristiche del grafo, in caso di score negativi verranno considerati nulli e quindi normalizzando un vettore contenente solo valori zero esso rimarrà identico. A l.43-46 viene calcolata la somma dei degree score dei moduli ‘Stot’ per poi poter normalizzare il vettore ‘PR’ (se uno degli score dei moduli è negativo, come già descritto, verrà considerato nullo (l.45)). Gli elementi del vettore ora vengono normalizzati: se lo score è positivo (l.48-49) la normalizzazione avviene regolarmente, se lo score è negativo (o nullo) il punteggio rimarrà inalterato e quindi sarà zero. In conclusione, la funzione restituisce il vettore ‘PR’.

Ora che è stato analizzato come il programma genera i moduli e gli assegna lo score, è necessario fare un passo indietro ed osservare lo pseudo-codice ad Imm.1. Si nota che ‘SimBioNeT’ aggiunge i moduli al network selezionandoli usando la funzione ‘sample’ (l.6). Questa funzione al suo interno genera i tre moduli utilizzando ‘createmod1’, ‘createmod2’ e ‘createmod3’ e ne calcola lo score utilizzando ‘score’. Si prosegue dunque analizzando la funzione ‘sample’, che genera un modulo per ognuna tipologia e ne seleziona solo uno in base allo score calcolato.

2.2.2 SELEZIONE DI UN MODULO DA AGGIUNGERE

```
5   def sample(G,V,Cref,gamma,od_id):
6       nodesavailable = len(V)
7       if nodesavailable == 1:
8           [M1, x, _] = createmod1(Cref,nodesavailable)
9           Mx = [[M1, x]]
10          M = [M1]
11      else:
12          [M1, x1, _] = createmod1(Cref,nodesavailable)
13          [M2, x2, _] = createmod2(Cref,nodesavailable)
14          [M3, x3, _] = createmod3(Cref,nodesavailable,gamma)
15          M = [M1, M2, M3]
16          Mx =[[M1, x1], [M2, x2], [M3, x3]]
17      [PR, Sij] = score(G,V,M,gamma,od_id)
18      nmod = len(M)
19      count = 0
20      while all(pr == 0 for pr in PR) and len(M) != 1:
21          if count > 10:
22              break
23          [M1, x1, _] = createmod1(Cref,nodesavailable)
24          [M2, x2, _] = createmod2(Cref,nodesavailable)
25          [M3, x3, _] = createmod3(Cref,nodesavailable,gamma)
26          M = [M1, M2, M3]
27          Mx =[[M1, x1], [M2, x2], [M3, x3]]
28          [PR, Sij] = score(G,V,M,gamma,od_id)
29          count += 1
30      if len(M) == 1:
31          choice = 0
32      else:
33          if all(pr == 0 for pr in PR):
34              choice = random.randint(0,nmod-1)
35          else:
36              choice = M.index(random.choices(M, PR)[0])
37      return(Mx[choice],Sij[choice])
```

Imm.18: codice della funzione ‘sample’

La funzione ‘sample’ (Imm.18) richiede in input ‘G’, ‘V’ e i coefficienti ‘Cref’, ‘gamma’ e ‘od_id’. Come prima cosa la funzione ottiene, attraverso la lunghezza di V, il numero di nodi ‘nodesavailable’ che possono essere connessi tra loro (l.6). Questo viene fatto perchè nel caso in cui ci sia un solo nodo disponibile può essere creato solamente il modulo1 a causa delle restrizioni di generazione degli altri due. Quindi in suddetto caso, a l.7-10 viene generato solamente il modulo1 attraverso ‘createmod1’ (l.8). Poi vengono create due liste: ‘Mx’ conterrà le coppie matrice di connessione-nodi regolatori per ogni modulo generato; ‘M’ conterrà solo le matrici di connessione per ogni modulo. In questa situazione, avendo generato solo il modulo uno, ovviamente le due liste avranno un solo elemento.
Negli altri casi vengono creati tutti e tre i moduli e salvati i risultati in ‘Mx’ e ‘M’ (l.12-16). Subito dopo la funzione calcola lo score per ogni modulo (l.17). Se i moduli generati risultano tutti avere score nullo, viene ripetuta la selezione nel tentativo di ottenere score positivi (l.20-29). In alcune situazioni però non è possibile ottenere dei moduli che abbiano degli score positivi, quindi, al decimo tentativo di selezione (conteggiato da ‘count’ inizializzato a zero a l.19), la selezione si blocca e vengono mantenuti i moduli generati dall’ultima esecuzione. In questo processo, a l.20 viene esclusa dal ciclo anche la situazione in cui si genera solo il modulo uno, in quanto non ha senso tentare un controllo per selezionare qualcosa di migliore quando la struttura generata può essere e sarà solamente una (un singolo nodo senza connessioni). Dopo aver selezionato un gruppo di moduli con score adeguati, la funzione seleziona uno tra essi da aggiungere al network. Viene cercato un valore ‘choice’ che rappresenta l’indice del modulo scelto dentro alla lista ‘Mx’. Ci sono tre situazioni possibili:

- nel caso in cui è stato generato solo il modulo uno viene assegnato in automatico ‘choice’ uguale a zero in quanto è l’unica scelta disponibile (l.30-31);
- se è stata raggiunta la decima generazione dei moduli e quindi hanno tutti score zero, ‘choice’ è ottenuta generando un numero intero casuale tra zero e il numero di moduli meno uno (l.33-34);
- in tutti gli altri casi ‘choice’ viene selezionata guardando i tre moduli e scegliendone uno con probabilità proporzionale agli score in ‘PR’ (l.36).

In conclusione, la funzione dà come output l’elemento di ‘Mx’ in posizione ‘choice’ e tutti gli score Sij ottenuti da ‘score’ per il modulo selezionato. Nel caso si volessero utilizzare dei moduli diversi da quelli generati dalle funzioni presenti nella libreria ‘modules’ bisognerebbe aggiornare manualmente questa funzione aggiungendo la generazione delle strutture aggiuntive (o rimuovendo alcuni dei tre moduli utilizzati nel caso studio), a l.12-16 e l.23-27 (e ovviamente anche ‘modules’). Inoltre, nel caso in cui non si utilizzino moduli che permettono la generazione con un singolo nodo, è necessario aggiornare l.8-10.

2.3 FORMAZIONE DELLA RETE E CALCOLO DEI RISULTATI

Una volta definiti tutti i meccanismi interni che permettono il funzionamento della simulazione si analizza il programma principale, la funzione 'SimBioNeT', che utilizzando quanto visto nei precedenti capitoli genera la struttura completa del network.

2.3.1 FORMAZIONE DELLA RETE

Il funzionamento di 'SimBioNeT' è già stato descritto dallo pseudo-codice ad Imm.1, quindi in questa sezione ci si sofferma solo su alcuni passaggi che non sono stati osservati in precedenza.

```
4   def SimBioNeT(N,Cref,gamma,od_id):
5       V = []
6       H = []
7       G = []
8       for i in range(N):
9           V.append(i)
10          G.append([False]*N)
11      while len(V) + len(H) != 1:
12          while len(V) != 0:
13              [[M,x],S] = sample(G,V,Cref,gamma,od_id)
14              v = len(V)
15              m = len(M)
16              Sj = [0]*m
17              for j in range(m):
18                  for i in range(v):
19                      if S[i][j] > 0:
20                          Sj[j] += S[i][j]
21              for i in range(v):
22                  for j in range(m):
23                      if S[i][j] > 0:
24                          S[i][j] = S[i][j]/Sj[j]
25                      else:
26                          S[i][j] = 0
27              nodestoconnect = []
28              for i in range(m):
29                  weights = [nodesV[i] for nodesV in S]
30                  if sum(weights) - sum(weights[k] for k, val in enumerate(V) if val in nodestoconnect) == 0:
31                      nodeselected = random.choice(V)
32                  else:
33                      nodeselected = random.choices(V, weights)[0]
34                  while nodeselected in nodestoconnect:
35                      if sum(weights) - sum(weights[k] for k, val in enumerate(V) if val in nodestoconnect) == 0:
36                          nodeselected = random.choice(V)
37                      else:
38                          nodeselected = random.choices(V, weights)[0]
39                  nodestoconnect.append(nodeselected)
40              for i in nodestoconnect:
41                  V.remove(i)
42              for i in range(m):
43                  for j in range(m):
44                      if M[i][j]:
45                          G[nodestoconnect[i]][nodestoconnect[j]] = True
46              for i in range(len(x)):
47                  H.append(nodestoconnect[i])
48          V = list(H)
49          H = []
50      return(G)
```

Imm.19: codice della funzione 'SimBioNeT'

La funzione a l.11 inizia con la formazione del network, il processo continua fino a che la somma del numero di nodi presenti in V e in H è uguale a uno, a significare che tutti i nodi sono stati connessi tra di loro ed è rimasto un singolo nodo a regolare tutti gli altri (l.11). Una volta ottenuto a l.13 un modulo da aggiungere al grafo, per collegarlo alla struttura preesistente in modo ottimale viene calcolato uno score che rappresenta quale tra i nodi i di 'V' ricopre il ruolo 'j' nel modulo in modo migliore, utilizzando il risultato 'S' di 'sample'. La funzione calcola lo score complessivo 'Sj' per

ogni j (escludendo i valori negativi) a l.17-20 e poi normalizza gli score Sij per ogni coppia (i,j) (l.21-26). Nel caso in cui si presenti uno score Sij negativo esso verrà portato automaticamente a zero, per assegnargli possibilità nulla di selezione. Una volta che per ogni j è stato calcolato uno score, per assegnare ai nodi di V i ruoli più adatti nel modulo, la funzione procede con la scelta di 'm' nodi in 'V'. Viene inizializzato un vettore 'nodestoconnect' (l.27) che conterrà tali nodi, poi la funzione procede con la selezione. Per ogni nodo del modulo vengono estratti da 'S' i pesi 'weights' assegnati ad ogni nodo di 'V' per la selezione di tale ruolo (l.29). A questo punto possono verificarsi due casistiche:

- se tutti gli score in 'weights' sono nulli vuol dire che nessuna selezione migliora il network, quindi verrà scelto un nodo a caso 'nodeselected' tra quelli possibili (l.30-31);
- negli altri casi verrà scelto un nodo tra quelli in 'V' con probabilità proporzionale a 'weights' (l.33).

Analizzando in maniera più puntuale la condizione espressa a l.30, la funzione non controlla solamente se la somma di tutti i punteggi per il ruolo nel modulo sia zero, ma lo fa per tutti i nodi disponibili. Questo perché se uno o più nodi all'interno di 'nodestoconnect' sono gli unici ad avere uno score positivo per quel ruolo e sono già stati scelti in iterazioni precedenti del ciclo, sarà impossibile fare una selezione. In quel caso, volendo entrare a l.31, senza questo controllo la funzione entrerebbe a l.33 e avverrebbe un loop infinito. Continuando nell'esecuzione del codice, se 'nodeselected' è già presente in 'nodetoconnect', la funzione svolge nuovamente la ricerca fino a che non viene scelto un nodo che non si trova in 'nodestoconnect' (l.34-38). Una volta selezionato un nodo adeguato, esso viene aggiunto a 'nodestoconnect' a l.39. Per il resto la funzione si conclude come già visto a capitolo 1.3 e restituisce in output 'G'.

2.3.2 CALCOLO DEI RISULTATI

Una volta generato il network, per poter fare delle considerazioni qualitative sui risultati ottenuti dal codice è necessario ricavare dal grafo generato i parametri essenziali definiti al capitolo 1.1. Per fare ciò si utilizza la funzione 'getfinalparameters' (Imm.20) subito dopo l'esecuzione di 'SimBioNeT'. Questa funzione permette di ricavare le distribuzioni del grado di connessione, il coefficiente medio di clustering e la cpl.

Le due funzioni vengono eseguite in sequenza all'interno del programma main, che chiede all'utente alcuni parametri in input essenziali per la formazione del network (numero di nodi 'N', 'Cref', 'gamma', 'od_id') e poi permette generazione del grafo e conseguente calcolo dei risultati.

```
5   def getfinalparameters(G):
6       m = len(G)
7       Pod = distrP(G,1)
8       Pid = distrP(G,2)
9       C = calcCm(m, calcK(G), calcq(G))
10      tot = 0
11      for i in range(m):
12          for j in range(i):
13              value = CPL(G,i,j)
14              tot += value
15      cpl = tot/sum(range(1,m))
16      return(Pid, Pod, C, cpl)
```

Imm.20: codice della funzione ‘getfinalparameters’

La funzione richiede come input solamente la matrice di connessione del grafo ‘G’. A l.7-8 viene calcolata la distribuzione out-degree e in-degree del grafo utilizzando ‘distrP’. A l.9 la funzione ottiene il coefficiente medio di clustering attraverso ‘calcCm’. A l.10-15 viene calcolata la cpl utilizzando la funzione ‘CPL’ che verrà descritta in seguito. Per ottenere questo parametro, la funzione calcola le distanze tra tutte le coppie di nodi e ne fa la media, evitando però di calcolare le distanze non necessarie. Infatti l’impostazione del ciclo a l.11-12, permette di selezionare solamente le coppie che nella matrice si trovano al di sotto della diagonale principale. In questo modo la funzione evita di calcolare tutte le distanze tra un nodo e se stesso (elementi sulla diagonale) e ottiene una volta sola la distanza tra le varie coppie di nodi. Se calcolasse le distanze per tutti gli elementi della matrice ad esclusione di quelli sulla diagonale, per ogni coppia di nodi (a,b) otterrebbe la distanza tra a e b e anche tra b e a, il che è una ripetizione inutile.

Come visto ad Imm.20, ‘getfinalparameters’ utilizza la funzione ‘CPL’ (Imm.21) per calcolare la distanza minima tra due nodi all’interno del grafo. Si procede ora descrivendola.

```
3   def CPL(G,a,b):
4       newconnext = []
5       nodes = list(range(len(G)))
6       nodes.remove(a)
7       newconn = obtainneighbors(G,a)
8       dist = 1
9       while b not in newconn:
10          dist += 1
11          for i in newconn:
12              nodes.remove(i)
13          for i in newconn:
14              add = (obtainneighbors(G,i))
15              for j in add:
16                  if j not in newconnext:
17                      newconnext.append(j)
18          toremove = []
19          for i in newconnext:
20              if i not in nodes:
21                  toremove.append(i)
22          for i in toremove:
23              newconnext.remove(i)
24          newconn = list(newconnext)
25          newconnext = []
26      return dist
```

Imm.21: codice della funzione ‘CPL’

La funzione richiede in input la matrice di connessione della struttura ‘G’ e gli indici ‘a’ e ‘b’ dei due nodi tra i quali si vuole calcolare la distanza. La funzione sfrutta ‘obtainneighbors’ per controllare se

tra i vicini di ‘a’ è presente ‘b’. Se così non fosse controlla se tra i vicini dei vicini di ‘a’ si trova ‘b’ e procede in questo modo fino a che non incontra ‘b’.
Viene creata una lista ‘nodes’ contenente tutti i nodi di ‘G’ (l.5) e subito rimosso ‘a’ da essa (l.6). Questa lista conterrà tutti i nodi del grafo che non sono ancora stati controllati dalla funzione. Viene fatto perchè una volta ottenuti i vicini di un nodo i e controllato se ‘b’ è presente tra essi, se i dovesse appartenere ai nodi da controllare di un’iterazione successiva non è necessario controllarlo di nuovo. A l.7 la funzione ottiene i vicini di ‘a’, poi inizia il ciclo che controlla se tra essi è presente ‘b’. Se questa condizione è vera, la funzione si conclude e dà come output la distanza ‘dist’, inizializzata a uno a l.8. Altrimenti, viene aggiornata la distanza a l.10 e rimossi da ‘nodes’ i vicini salvati in ‘newconn’ (l.11-12). Poi per ogni nodo presente in ‘newconn’ vengono cercati i vicini e aggiunti ad un vettore ‘newconnext’ (l.13-17). Dentro a ‘newconnext’ è necessario che ci siano solo nodi univoci, quindi ogni volta che la funzione tenta di aggiungere un nodo al vettore, viene prima controllato che esso non sia già presente (l.15-17). Subito dopo la funzione guarda quali tra i nodi in ‘newconnext’ non si trovano dentro a ‘nodes’ (se questo è vero significa che il nodo è già stato analizzato in un’iterazione passata) e li aggiunge ad un vettore ‘toremove’ (l.18-21). Poi ogni elemento presente in ‘toremove’ viene rimosso da ‘newconnext’ (l.22-23), viene posto ‘newconn’ uguale a ‘newconnext’ (l.24) e svuotato ‘newconnext’ (l.25), poi il ciclo while riparte. In conclusione, la funzione restituisce in output ‘dist’.

A questo punto è conclusa la generazione del network e il calcolo dei risultati ad esso associati. Questo processo dovrebbe portare all’ottenimento di strutture con distribuzioni in e out-degree coerenti con la legge power-law ad Eq.1 per il parametro ‘gamma’ inserito in input, un coefficiente medio di clustering vicino a quello dato in input e ad un grafo che presenta comportamento small world. Nel prossimo capitolo verranno fatte alcune considerazioni sui risultati ottenuti dall’esecuzione del codice.

3. RISULTATI

Si procede eseguendo il programma main numerose volte per osservare i risultati ottenuti e valutare se i grafi generati sono compatibili con le richieste al capitolo 1.1.

3.1 CALCOLO DELL'ERRORE

Serve dunque controllare che la distribuzione del grado di connessione segua la power-law ad Eq.1, il coefficiente medio di clustering resti costante nel tempo e che il network abbia comportamento small world. Per quanto riguarda la modularità il modello MTM la prevede implicitamente.
Per verificare la qualità della distribuzione del grado di connessione, si utilizza il coefficiente R-squared, che ha espressione:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(Y_i - y_i)^2}{\sum_{i=1}^{n}(Y_i - \bar{Y})^2}$$

Eq.8: definizione del parametro R-squared

in cui 'Y' rappresenta la distribuzione desiderata e definita da Eq.1, 'y' quella ottenuta dalla simulazione e $\bar{Y}$ la media tra i valori di Y. Questo coefficiente esprime se i dati del network generato rappresentano in modo adeguato la distribuzione desiderata. Il valore di R-squared è compreso tra zero e uno, maggiore è il risultato, più il grafo ha distribuzione del grado di connessione spiegabile attraverso la power-law. Se R-squared è uguale a uno significa che i dati simulati sono perfettamente rappresentati dalla power-law, se è pari a zero vuol dire che il fit è inesistente.

3.2 CONSIDERAZIONI SUI TEMPI DI ESECUZIONE

Tutti i test sono stati eseguiti su un computer portatile con processore Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz 1.80 GHz; 8GB RAM con velocità 2133MHz.
L'obiettivo del progetto è stato quello di scrivere un codice funzionante, che permettesse la creazione di grafi che rispettassero le richieste di un network trascrizionale. Non ci si è dunque soffermati con particolare attenzione sull'ottimizzazione dei processi e la riduzione dei tempi di esecuzione. Nel codice che crea i grafi sono stati ottimizzati alcuni passaggi e complessivamente il tempo di esecuzione è buono. Per generare un network con N=100 in media il computer impiega 0.5 secondi, per N=1000 circa 165 secondi, meno di tre minuti. Questi valori non comportano nessun problema, tuttavia durante il calcolo dei parametri ricercati, il tempo di esecuzione aumenta vertiginosamente. Questo perché il codice che ottiene la cpl (calcolo in ogni caso lungo e complesso per network di grandi dimensioni) non è stata ottimizzata al massimo e utilizza una strategia quasi brute force, evitando solamente analisi già fatte (Imm.21, nodi che sono già stati analizzati non vengono mai più osservati) ed eseguendo solo i controlli necessari (Imm.21, si calcolano le distanze tra nodi in una

sola direzione), ma comunque il numero di distanze tra nodi da calcolare in network con N elevato sono moltissime.

N	Numero di distanze tra nodi per network da calcolare	Tempo di esecuzione complessivo per 20 network
100	4950	65 secondi
200	19900	15 minuti
300	44850	75 minuti
500	124750	9 ore
1000	499500	136 ore

Tab.4: numero di controlli e tempistiche di esecuzione del main per diversi valori di N

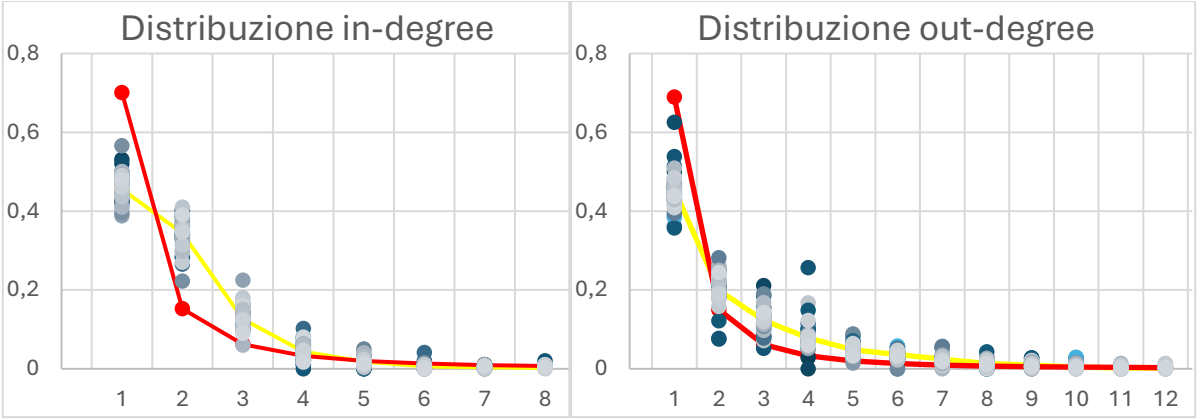
A Tab.4 sono descritte le tempistiche di esecuzione del programma main per la generazione di venti network (per N=500 e N=1000 sono dati stimati basandosi sui precedenti). Dunque, il calcolo della cpl necessita un'enorme quantità di tempo, rappresentando quasi interamente la durata dell'esecuzione del programma. Per questo motivo per network con N>300 sono stati fatti dei test senza il calcolo della cpl, perché non è detto che il computer sarebbe riuscito a gestire l'esecuzione del programma che in ogni caso avrebbe richiesto tempi troppo elevati.

3.3 TEST ESEGUITI E RISULTATI OTTENUTI

Si utilizza il programma main per simulare dei grafi e visualizzare i risultati ottenuti. Per verificare che le proprietà richieste al capitolo 1.1 siano rispettate, è necessario testare la generazione utilizzando diversi set di parametri. Si ricorda che il programma main in input richiede i valori 'N', 'gamma', 'Cref' e 'od_id'. Per verificare se il grado di connessione rispetta la power-law ad Eq.1 con il valore di gamma assegnato in input, bisogna controllare questo dato sia per la generazione del network basata sulla distribuzione out-degree (od_id=1) che per la in-degree (od_id=2). Per avere coefficiente medio di clustering costante con l'aumento del numero di nodi del grafo è necessario generare network con numero di nodi diverso e confrontare i risultati. Anche per osservare il comportamento small-world si può lavorare in questo modo.

Si procede generando grafi a gruppi di venti per dataset fissati. Nei seguenti test verranno sempre utilizzati gamma=2.2 e Cref=0.4, gli altri due parametri invece verranno modificati. Si creano grafici con numero variabile di nodi, prima utilizzando od_id=1 e poi od_id=2. Come N si utilizzano i seguenti valori: 50, 100, 200, 300, 500, 750. Come detto al capitolo 3.2, per N>300 i test sono stati eseguiti senza il calcolo della cpl.

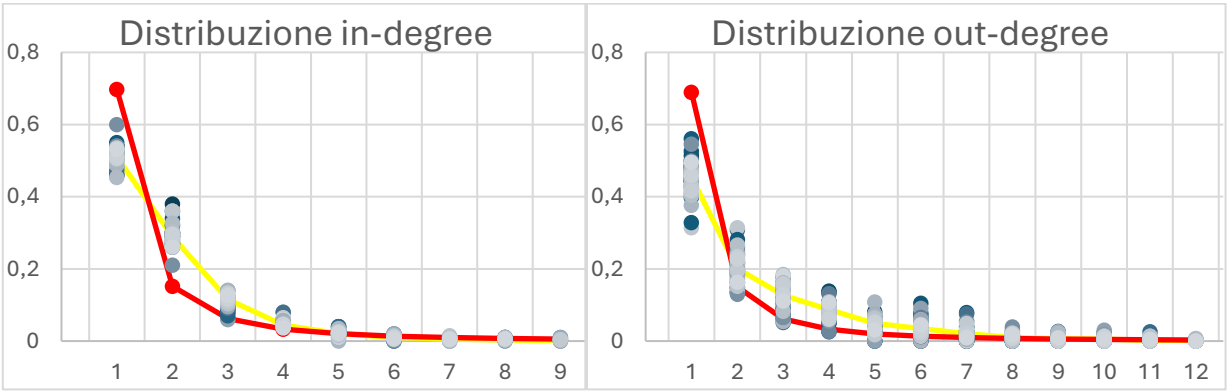
Si osserva la distribuzione del grado di connessione ottenuta per od_id=1; Cref=0.4; gamma=2.2 e N variabile come appena descritto. Effettuando la generazione basata sulla out-degree si hanno i risultati rappresentati ad Imm.22.



Imm.22: distribuzioni in-degree e out-degree ottenute dalla simulazione di grafi basata sulla out-degree. La linea rossa rappresenta la distribuzione desiderata, quella gialla la distribuzione media ottenuta

I valori di R-squared per la distribuzione in-degree e la out-degree sono rispettivamente 0.744 e 0.846. Per quanto riguarda la out-degree il modello approssima in modo buono i dati richiesti, la distribuzione del grado di connessione è esprimibile attraverso una power law in cui il parametro gamma non è esattamente 2.2 come immesso in input (si ha R-squared=0.987 per gamma=1.4) ma ne è una buona approssimazione. Per quanto riguarda la in-degree invece, il modello non riesce esattamente ad approssimare la power-law, anche se comunque il risultato ottenuto non è negativo. In ogni caso questo dato va interpretato in maniera differente, perché è stato osservato[5] che per network trascrizionali la distribuzione del grado di connessione per la in-degree non è sempre descrivibile con una power-law, ma spesso ha andamento esponenziale.

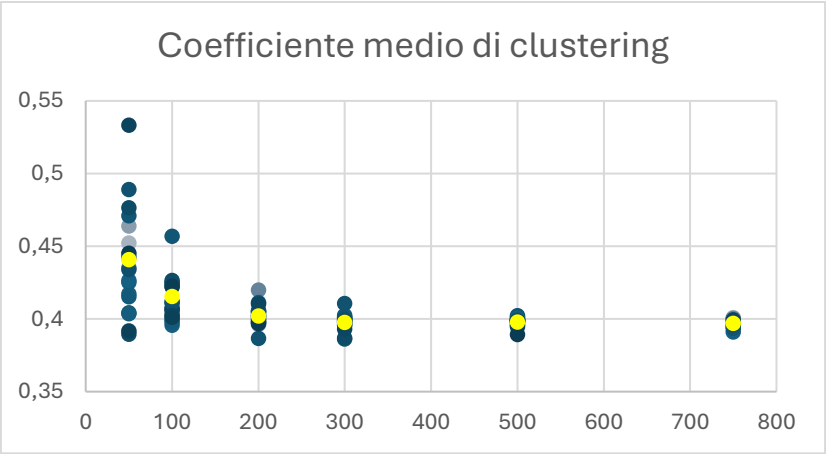
Mantendo il precedente dataset e cambiando solamente od_id=2, si ottengono le distribuzioni a Imm.23.



Imm.23: distribuzioni in-degree e out-degree ottenute dalla simulazione di grafi basata sulla in-degree. La linea rossa rappresenta la distribuzione desiderata, quella gialla la distribuzione media ottenuta

In questo caso i valori di R-squared ottenuti sono 0.859 (in-degree) e 0.841 (out-degree). Si osserva che generando i network basandosi sulla in-degree i risultati ottenuti per quanto riguarda tale distribuzione sono descrivibili dalla power-law in modo più adeguato, anche se a causa dei valori ottenuti per il grado di connessione K=2 non è possibile ottenere un fit perfetto. Il valore di R-squared più alto ottenibile è 0.931, per gamma=1.6. Invece, per quanto riguarda la distribuzione out-degree, i risultati ottenuti sono molto simili alla generazione precedente, confermando che anche con simulazioni in cui la out-degree non è controllata, durante la creazione del network la distribuzione ottenuta è comunque coerente con una legge power-law.

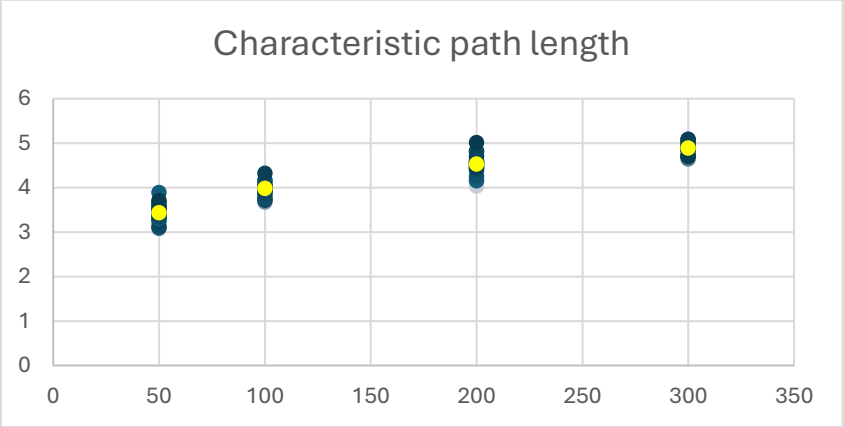
Per quanto riguarda i valori del coefficiente di clustering e della characteristic path length si osservano i risultati ottenuti variando il numero di nodi del network e mettendoli a confronto (Imm.24 e Imm.25). Si considera quanto ottenuto dalle generazioni con od_id=1 e od_id=2 insieme perché i risultati nei due casi sono pressoché identici.



Imm.24: coefficienti medi di clustering ottenuti per simulazioni con vari valori di N. Il valore in giallo rappresenta il valore medio

Per quanto riguarda il coefficiente medio di clustering, il programma funziona in modo tale da ottenere come risultato il valore 'Cref' inserito in input. Dunque, si osserva che i valori ricavati si trovano tutti attorno 'Cref', impostato inizialmente a 0.4 e hanno dispersione molto bassa. L'unico caso per il quale si ha una dispersione più consistente (e anche un risultato peggiore, 0.441) è per N=50. Questo risultato è probabilmente dovuto al basso numero di moduli utilizzati per generare il network. Avendo poche strutture interconnesse tra loro il programma fa più fatica a stabilizzare il coefficiente di clustering. Questo può essere confermato dai risultati ottenuti per valori di N più alti. Infatti, per N>50 hanno dispersione bassa e il coefficiente di clustering medio si stabilizza e risulta molto vicino a 0.4. Si può dire che C resta costante all'aumentare di N, in quanto tra N=200 e N=750 il valore medio varia di -0.005, in modulo solo l'1.24%. L'unica variazione più netta come già

evidenziato è presente tra N=50 e N=100 (5.89%) e più lieve tra N=100 e N=200 (3.13%), anche questa dovuta alla stabilizzazione spiegata in precedenza.



Imm.25: cpl ottenuta per simulazioni con vari valori di N. Il valore in giallo rappresenta il valore medio

Osservando la cpl invece, si osserva una crescita nel suo valore con l'aumento di N e dispersione dei risultati abbastanza costante. In tutti i network generati si osserva sempre comportamento small world. Il valore medio di cpl aumenta con N, ma tende a stabilizzarsi attorno a 5. Anche questo risultato probabilmente è dovuto al valore di coefficiente di clustering più elevato ottenuto per N bassi. Infatti, avendo C più alto, i nodi saranno maggiormente interconnessi tra loro e di conseguenza i percorsi effettuati per passare da un nodo all'altro saranno più corti.

Simulando network per N>300 sicuramente sarebbe stato possibile ottenere dei dati statistici utili, ma purtroppo, come spiegato al capitolo 3.2, non è stato possibile.

4. CONCLUSIONE

Pur non rappresentando in maniera perfetta il modello richiesto, il codice scritto genera dei network che ne sono una buona approssimazione. Con ulteriori analisi del codice e ottimizzazioni per quanto riguarda la performance, il codice può sicuramente arrivare a generare grafi che raggiungano le specifiche dei network trascrizionali in maniera più precisa. Tra le varie cose, si è visto che sicuramente è necessario rendere più efficiente la funzione 'CPL' per ottenere dei tempi di esecuzione più gestibili.

Una riflessione può essere fatta sulla tipologia di moduli utilizzati. Il codice è stato scritto in modo tale da poter usare anche altre strutture modulari per generare i grafi, questa funzionalità può essere sfruttata per creare network basati su moduli di vario tipo. Quindi preso un organismo, una tipologia di cellula o qualsiasi tipo di vivente, se è possibile identificare delle strutture modulati all'interno dei suoi network trascrizionali, SimBioNeT permette la simulazione di grafi per quello specifico organismo. I moduli utilizzati nel codice descritto al capitolo 2 sono le tre strutture maggiormente incontrate nei network di E.coli[8], ma questo processo può essere fatto con qualsiasi organismo, con la restrizione che i suoi network rispettino le regole viste al capitolo 1.1.

Un altro utilizzo che può avere questo simulatore è osservare come la presenza di particolari strutture modulari influenzino i parametri di un network. Ad esempio, per il caso preso in esame sono stati eseguiti dei test aumentando il numero massimo di nodi possibile all'interno del modulo di tipologia uno e si è osservato che i risultati ottenuti sono peggiori.

In studi più recenti è stata utilizzata la simulazione di grafi interconnessi (attraverso meccanismi più complessi) per determinare i percorsi di collegamento tra malattie di diverso tipo[20]; la connessione tra varie tipologie della stessa malattia ed elementi in comune[19]; collegamento tra marcatori neurochimici e comportamenti umani incontrati in un caso particolare[18].

Negli anni successivi alla pubblicazione di SimBioNeT la ricerca in questo campo si è estesa e altri simulatori sono stati progettati, uno tra questi è FRANK[21], progettato nel 2017 da Clément Carré, André Mas e Gabriel Krouk. Anch'esso genera network basandosi sui dati noti di E.coli, ma tali strutture sono più complesse e con un numero molto maggiore di nodi (fino a 10000) e vengono ottenute con un tempo di computazione molto basso rispetto alla grandezza del grafo.

5. BIBLIOGRAFIA

1. Di Camillo, B., Toffolo, G. & Cobelli, C. A Gene Network Simulator to Assess Reverse Engineering Algorithms. Annals of the New York Academy of Sciences 1158, 125–142 (2009).

2. Di Camillo, B., Falda, M., Toffolo, G. & Cobelli, C. SimBioNeT: A Simulator of Biological Network Topology. IEEE/ACM Trans. Comput. Biol. and Bioinf. 9, 592–600 (2012).

3. Jeong, H., Tombor, B., Albert, R. & Oltvai, Z. N. The large-scale organization of metabolic networks. (2000).

4. Featherstone, D. E. & Broadie, K. Wrestling with pleiotropy: Genomic and topological analysis of the yeast gene expression network. BioEssays 24, 267–274 (2002).

5. Babu, M. M., Luscombe, N. M., Aravind, L., Gerstein, M. & Teichmann, S. A. Structure and evolution of transcriptional regulatory networks. Current Opinion in Structural Biology 14, 283–291 (2004).

6. Wagner, A. & Fell, D. A. The small world inside large metabolic networks. Proc. R. Soc. Lond. B 268, 1803–1810 (2001).

7. Milo, R. et al. Network Motifs: Simple Building Blocks of Complex Networks.

8 Shen-Orr, S. S., Milo, R., Mangan, S. & Alon, U. Network motifs in the transcriptional regulation network of Escherichia coli. Nat Genet 31, 64–68 (2002).

8. Ravasz, E., Somera, A. L., Mongru, D. A., Oltvai, Z. N. & Barabási, A.-L. Hierarchical Organization of Modularity in Metabolic Networks. Science 297, 1551–1555 (2002).

9. Watts, D. J. & Strogatz, S. H. Collective dynamics of ‘small-world’ networks. (1998).

10. Kauffman SA. Metabolic stability and epigenesis in randomly constructed genetic nets. J Theor Biol. 22(3):437-67 ($1^9$69).

11. Glass, L. & Hill, C. Ordered and disordered dynamics in random networks. Europhys. Lett. 41, 599–604 (1998).

12 Barabasi, A.-L. & Albert, R. Emergence of Scaling in Random Networks.

13. Pržulj, N., Corneil, D. G. & Jurisica, I. Modeling interactome: scale-free or geometric? Bioinformatics 20, 3508–3515 (2004).

14. Zhao, J., Yu, H., Luo, J.-H., Cao, Z.-W. & Li, Y.-X. Hierarchical modularity of nested bow-ties in metabolic networks. BMC Bioinformatics 7, 386 (2006).

15. Hartwell, L. H., Hopfield, J. J., Leibler, S. & Murray, A. W. From molecular to modular cell biology. Nature 402, C47–C52 (1999).

16. Lee, T. I. et al. Transcriptional Regulatory Networks in Saccharomyces cerevisiae. Science 298, 799–804 (2002).

17. Hleap, J., Susko, E. & Blouin, C. Defining structural and evolutionary modules in proteins: a community detection approach to explore sub-domain architecture. BMC Struct Biol 13, 20 (2013).

18. Chacko, T. P. et al. A regulatory pathway model of neuropsychological disruption in Havana syndrome. Front. Psychiatry 14, 1180929 (2023).

19. Ji, J. et al. Detection for pathway effect contributing to disease in systems epidemiology with a case–control design. BMJ Open 5, e006721 (2015).

20. Chacko, T. P. et al. Mapping the network biology of metabolic response to stress in posttraumatic stress disorder and obesity. Front. Psychol. 13, 941019 (2022).

21. Carré, C., Mas, A. & Krouk, G. Reverse engineering highlights potential principles of large gene regulatory network design and learning. npj Syst Biol Appl 3, 17 (2017).