Università di Perugia
Dipartimento di Matematica e Informatica

Artificial Intelligent Systems: Intelligent Models

Social Graph Formation Model

Professore
Prof. Alfredo Milani

Studente
Tommaso Romani

Anno Accademico 2022-2023

Indice

1 Introduzione **3**
1.1 Richiami Teorici 3
1.2 Tecniche di implementazione 4

2 Implementazione **5**
2.1 main.py 5
2.2 run.py 7
2.3 link_pred.py 8
2.4 pref_attachment.py 11
2.5 metrics.py 13
2.5.1 Diameter 14
2.5.2 Clustering Coefficient 15
2.5.3 Maximum Connected Component 16
2.5.4 Average Shortest Path Lenght 17
2.6 Output 18

3 Analisi dei Costi **20**
3.1 Costo Generazione Rete 20
3.2 Costo Calcolo Metriche 21
3.2.1 Diametro 21
3.2.2 Clustering 22
3.2.3 Maximum Connected Components 22
3.2.4 Average Path Length 22

4 Analisi dei Risultati **23**
4.1 Nodes degree distribution 24
4.2 Clustering Coefficient Distribution 25
4.3 Osservazioni 26

5 Conclusioni **26**

1 Introduzione

Questo progetto prevede la creazione e l'analisi di un **Graph Formation Model** che va a simulare la formazione di una rete sociale.

La formazione del grafo deve rispettare le seguenti regole:

> Dato un insieme di nodi N di cui 4 sono inizialmente collegati tra di loro, e K iterazioni, ad ogni iterazione un nodo x viene estratto tra i N nodi con probabilità uniforme.
>
> Ad ogni nodo x estratto viene aggiunto un link nei seguenti modi:
>
> - Con una tecnica di link prediction con probabilità $p = \frac{|\Gamma(x)}{|\Gamma(x)|=c}$: per scegliere il nodo da connettere ad x si applica l'algoritmo di link prediction **Adamic Adar** a tutti i nodi non connessi a distanza 2 da x.
> I nodi sono classificati in base all'indice di **Adamic Adar** e il link di rango massimo viene aggiunto. Se non esiste nodo tale da poter applicare l'algoritmo allora si aggiunge un link per **Preferential attatchment**.
> - Per **Preferntial attachment** con probabilità $1 - p$ di scegliere il nodo da collegare a x
>
> Il sistema deve consentire di:
>
> 1. Configurare i parametri e di sperimentare diversi valori di c, con test sia per $0 < c \leq 1$ e $c > 1$, diversi valori di N e anche di K.
> 2. Di salvare la rete creata in `.csv` per poter comparare varie metriche calcolate su diverse simulazioni, *degree distribution* al variare del numero delle iterazioni, *max connected* components, *network diameter*, *average path*, *clustering coefficient*, e *average node connectivity*.

Tutto il codice può essere trovato al seguente link: Social Graph Formation Model Project

1.1 Richiami Teorici

Dato che questa rete tende a collegare i link poco connessi tramite preferential attatchment probabilmente viene ben modellata dal modello di **Barbasi-Albert**.

Questo è giustificato sia dal fatto che la rete simula una crescita, dato che nello step iniziale abbiamo $N-4$ nodi isolati, ed ad ogni iterazione si crea un collegamento con uno di questi nodi, che dal fatto che come già affermato, i nodi di bassa cardinalità cercano di creare collegamenti tramite *preferential attachment*.

Ciò significa che possiamo aspettarci l'emersione di **Hubs**(componenti estremamente connesse), e una *degree distribution* che segua la **Power Law**.

Per quanto riguarda la tencnica di link prediction *Adamic-Adar*, essa si basa sulla seguente formula:

$$A(x,y) = \sum_{u \in N(x) \cap N(y)} \frac{1}{|log(N(u))|}$$

Questo approccio predice link osservando i vicini comuni tra 2 nodi, nel nostro caso quelli a distanza 2, e assegnando loro uno score che è inversamente proporzionale al numero di collegamenti presenti su questi vicini comuni.

Questo va a significare che vicini comuni con pochi altri collegamenti sono più significativi di vicini comuni con tanti collegamenti per la formazione di nuovi collegamenti.

1.2 Tecniche di implementazione

Per l'implementazione di questo progetto ho utilizzato **Pyton 3.12** che mi ha permesso di usufruire della libreria di network analisys **NetworkX** oltre ad alcune librerie grafiche per poter realizzare una visualizzazione interattiva dell'evoluzione della rete, in particolare **Plotly**.
Le analisi statistiche finali delle varie metriche sono state effettuate utilizzando la libreria **Pandas**.

Figura 1: Librerie utilizzate per la realizzazione del progetto

2 Implementazione

Per l'Implementazione ho suddiviso il progetto in vari file:`main.py`, `run.py`, `pref_attatchment.py`, `link_pred.py`, `metrics.py`.

2.1 main.py

Nel file `main.py` viene inizializzato il grafo e fatta partire la simulazione.
In particolare usando `networkx` il grafo viene inizializzato nel modo seguente:

```python
# Function that given initial parameters, initializes the starting graph
def initializer(n: int):
    # Initialize graph
    G = nx.Graph()

    # Add nodes
    for i in range(n):
        G.add_node(i)
    # Add edges randomly to 4 nodes of the starting graph
    G = add_random_edges(G, random.randint(4, 16))

    return G
```

La funzione `inizializer()` inizializza un oggetto `nx.Graph()` a cui vengono aggiunti N nodi e poi tramite `add_random_edges`

```python
def add_random_edges(G, num_edges=8):

    # Ensure there are enough nodes in the graph
    if G.number_of_nodes() < 4:
        raise ValueError("Graph must contain at least 4 nodes")

    # Convert NodeView to list and select 4 unique nodes
    nodes = random.sample(list(G.nodes()), 4)
    print(len(nodes))
    # Add edges between the 4 nodes randomly
    for i in range(num_edges):
        index = random.randint(0,3)
        G.add_edge(nodes[index], nodes[(index+random.randint(1,3))%4])

    return G
```

La funzione `add_random_edges()` prima controlla che il numero dei nodi sia sufficientemente alto per selezionare 4 nodi da collegare in modo random, e poi aggiunge `num_edges` link tra i 4 nodi selezionati precedentemente.

Tutti gli argomenti possono essere passati alla funzione grazie ad `argparse`, che grazie al parametro `input` permette di caricare un grafo precedentemente generato e salvato in `.csv`, altrimenti se non viene passato nessun file esegue normalmente la simulazione.

```
# Initialize graph
G = initializer(args.n)
# Draw graph
print(G)

# Iterate over graph
G = iterate_and_animate(G, args.k, args.c)
# Save graph
save_graph(G,
           degreeCount,
           max_cc,
           diameter,
           clustering_coefficient,
           args.output)
print(G)
```

Ora andremo ad analizzare le varie funzioni chiamate da questa parte di codice.

2.2 run.py

All'interno di `run.py` abbiamo le funzioni che eseguono la logica principale, più alcune funzioni di utility, come ad esempio quella che salva la rete su un file `.csv` e quella che estrae la probabilità p definita nella descrizione del problema.

La funzione `iterate_and_animate()` itera K volte sul grafo, e ad ogni iterazione genera prima un frame per la visualizzazione finale dell'iterazione attuale, e succes-

sivamente chiama la funzione `graph_iterator()`, che va ad aggiungere collegamenti alla rete.

```
def graph_iterator(G:nx.Graph, c:int):

    # Select a random node
    node = random.choice(list(G.nodes()))
    p = probability(G, node, c)

    if random.random() < p:
        #computese adamic adar index
        G = adamic_adar(G, node)
    else:
        #adds link by preferential attachment
        G = preferential_attachment(G, node)

    return G
```

A seconda della probabilità p ottenuta tramite la funzione `probability()` che verrà mostrata di seguito, l'algoritmo decide se aggiungere un link tramite *Adamic Adar* o tramite *Preferential attatchment*.

```
def probability(G:nx.Graph, node:int, c:int):
    p = G.degree(node)/(G.degree(node)+c)
    return p
```

2.3 link_pred.py

All'interno di `link_pred.py` è implementato l'algoritmo che calcola il valore di *Adami Adar* per tutti i vicini a distanza 2 del nodo estratto.

Il codice per la sua implementazione è il seguente:

```python
def adamic_adar(G, node):
    # Initialize a set to store the nodes at distance two
    neighbours  = set()
    # Perform BFS from the start node
    for nodes, distance in nx.single_source_shortest_path_length(G, node).items():
        # If the distance is two, add the node to the set
        if distance == 2:
            neighbours.add(nodes)

    maximum = None
    for nodes in neighbours:
        if len(neighbours) == 0:
            # adds link trough preferential attachment
            G = preferential_attachment(G, node)
            return G
        else:
            # Compute the Adamic-Adar index
            try:
                score = sum(1 / math.log(G.degree(v)) for v in G[nodes])
            except ZeroDivisionError:
                # adds link trough preferential attachment
                G = preferential_attachment(G, node)
                return G
            # Add the score as a node attribute
            G.nodes[nodes]['score'] = score
            # Update maximum if it's None or if the current
            # node has a higher score
            if maximum is None or G.nodes[nodes]['score']
                                    > G.nodes[maximum]['score']:
                maximum = nodes
        G.add_edge(node, maximum)
    # Return node with maximum score
    return G
```

All'inizio cerca tutti i vicini a distanza 2 con una `BFS` a partire dal nodo interessato, salvandoli nel set `neighbours`.

```
    # Initialize a set to store the nodes at distance two
    neighbours  = set()
    # Perform BFS from the start node
    for nodes, distance in nx.single_source_shortest_path_length(G, node).items():
        # If the distance is two, add the node to the set
        if distance == 2:
            neighbours.add(nodes)
```

Successivamente controlla se `neighbours` è vuota, e in caso positivo cerca la nuova connessione usando il *preferential attatchment*, altrimenti si calcola il coefficiente di *Adamic-Adar*. Poi dopo averlo calcolato per ogni vicino trovato seleziona il link di valore massimo, che corrisponde a quello più probabile.

```
maximum = None

if len(neighbours) == 0:
    # adds link trough preferential attachment
    G = preferential_attachment(G, node)
    return G
else:
    for nodes in neighbours:
        # Compute the Adamic-Adar index
        try:
            score = sum(1 / math.log(G.degree(v)) for v in G[nodes])
        except ZeroDivisionError:
            # adds link trough preferential attachment
            G = preferential_attachment(G, node)
            return G
        # Add the score as a node attribute
        G.nodes[nodes]['score'] = score
        # Update maximum if it's None or if the current
        # node has a higher score
        if maximum is None or G.nodes[nodes]['score']
                                > G.nodes[maximum]['score']:
            maximum = nodes
    G.add_edge(node, maximum)
# Return node with maximum score
return G
```

2.4 pref_attachment.py

In questo file è implementata la funzione che crea un nuovo collegamento tramite *preferential attatchment*.

```
def preferential_attachment(G, node):
# Get all other nodes in the graph

    nodes = [n for n in G.nodes() if n != node]

    # Get all combinations of the given node with the other nodes
    combinations = [(node, n) for n in nodes]

    for i, j in combinations:
        # Calculate the PA score
        G.nodes[j]['score'] = (G.degree(i) + 1) * (G.degree(j)+ 1)
        #print(G.nodes[j]['score'])

    # Stocastically select a node to add the edge to based on the PA score
    j = random.choices(nodes,
                       weights=[G.nodes[n]['score'] for n in nodes],
                       k=1)[0]
    # Add the edge to the graph
    G.add_edge(node, j)

    return G
```

Inizialmente genera la combinazione del nodo che deve essere collegato con tutti gli altri nodi del grafo, e successivamente si calcola il *Preferential Attatchment score* per ogni possibile combinazione.

```
nodes = [n for n in G.nodes() if n != node]

# Get all combinations of the given node with the other nodes
combinations = [(node, n) for n in nodes]

for i, j in combinations:
    # Calculate the PA score
    G.nodes[j]['score'] = (G.degree(i) + 1) * (G.degree(j)+ 1)
    #print(G.nodes[j]['score'])
```

Successivamente usando la libreria `random` di python estrae stocasticamente un nodo con cui effettuare il collegamento usando il **PA score** come peso per dare precedenza ai nodi che hanno già molti collegamenti.

```
    # Stocastically select a node to add the edge to based on the PA score
    j = random.choices(nodes,
                       weights=[G.nodes[n]['score'] for n in nodes],
                       k=1)[0]
    # Add the edge to the graph
    G.add_edge(node, j)

    return G
```

2.5 metrics.py

Nel file `mterics.py` sono presenti i vari metodi per il calcolo delle metriche, che ora andremo a analizzare nel dettaglio:

2.5.1 Diameter

```
def network_diameter(G: nx.Graph):
max_path_length = 0
for source in G.nodes():
    distances = {node: float('inf') for node in G.nodes()}
    distances[source] = 0
    queue = [source]
    while queue:
        current_node = queue.pop(0)
        for neighbor in G.neighbors(current_node):
            if distances[neighbor] == float('inf'):
                distances[neighbor] = distances[current_node] + 1
                queue.append(neighbor)
                max_path_length = max(max_path_length, distances[neighbor])
return max_path_length
```

Questa funzione calcola il diametro del grafo iterando su ogni nodo del grafo, inizializzando per ogni nodo un dizionario `distances` che è inizialmente inizializzato a infinito per tutti i nodi diversi da `source` per cui è inizializzato a 0.
Successivamente viene creata una coda `queue` per tener traccia dei nodi visitati e, fino a quando `queue` non è vuota itera controllando se partendo dal nodo attuale sono presenti vicini a distanza ∞, che quindi ancora non sono stati visitati. In caso positivo aggiorna la loro distanza con quella nel noto attuale $+1$, aggiunge tale nodo alla `queue` e aggiorna `max_path_length` se il valore attuale è maggiore del massimo trovato fino ad ora.

Dopo aver visitato tutti i nodo `max_path_length` corrisponderà al diametro della rete.

2.5.2 Clustering Coefficient

```
def node_clustering_coefficient(G: nx.Graph):
    clustering_coefficients = {}
    for node in G.nodes():
        neighbors = list(G.neighbors(node))
        num_neighbors = len(neighbors)
        if num_neighbors < 2:
            clustering_coefficients[node] = 0.0
        else:
            num_triangles = 0
            for i in range(num_neighbors):
                for j in range(i+1, num_neighbors):
                    if G.has_edge(neighbors[i], neighbors[j]):
                        num_triangles += 1
            clustering_coefficients[node] = 2.0 * num_triangles / (num_neighbors * (n
    return clustering_coefficients
```

La funzione `node_clustering_coefficient()` calcola il clustering coefficient per ogni nodo presente nella rete.
Come prima cosa inizializza un dizionario vuoto `clustering_coefficients` che salverà i coefficienti di clustering per ogni nodo.
La funzione itera su ogni nodo della rete, e per ogniuno si genera una lista dei propri vicini. Se ha meno di 2 vicino non si possono formare triangoli e quindi il coefficiente di clustering sarà 0, altrimenti la funzione calcola il numero di triangoli di cui il nodo fa parte.
Infine usa questo valore per calcolarsi il clustering coefficient, dividendolo per il numero totali di connessioni possibili dei suoi vicini.

2.5.3 Maximum Connected Component

```
def maximum_connected_components(G:nx.Graph):
    visited = set()
    max_cc = []
    for node in G.nodes():
        if node not in visited:
            cc = []
            stack = [node]
            while stack:
                current_node = stack.pop()
                if current_node not in visited:
                    cc.append(current_node)
                    visited.add(current_node)
                    stack.extend(G.neighbors(current_node))
            if len(cc) > len(max_cc):
                max_cc = cc
    return max_cc
```

La funzione inizialmente inizializza un insieme vuoto `visited` per tenere traccia dei nodi già visitati e un elenco vuoto `max_cc` per memorizzare i nodi nella componente connessa più grande trovata finora, successivamente itera su tutti i nodi del grafo, e per ogni nodo inizia una ricerca in profondità a partire da quel nodo per vedere quali nodi appartengono alla stessa componente connessa.

Dopo aver terminato la ricerca in profondità controlla se la dimensione di `cc` è maggiore di `max_cc`, in caso positivo ha trovato una componente connessa di dimensione maggiore rispetto a quella precedente, e quindi la sovrascrive con quella attuale.

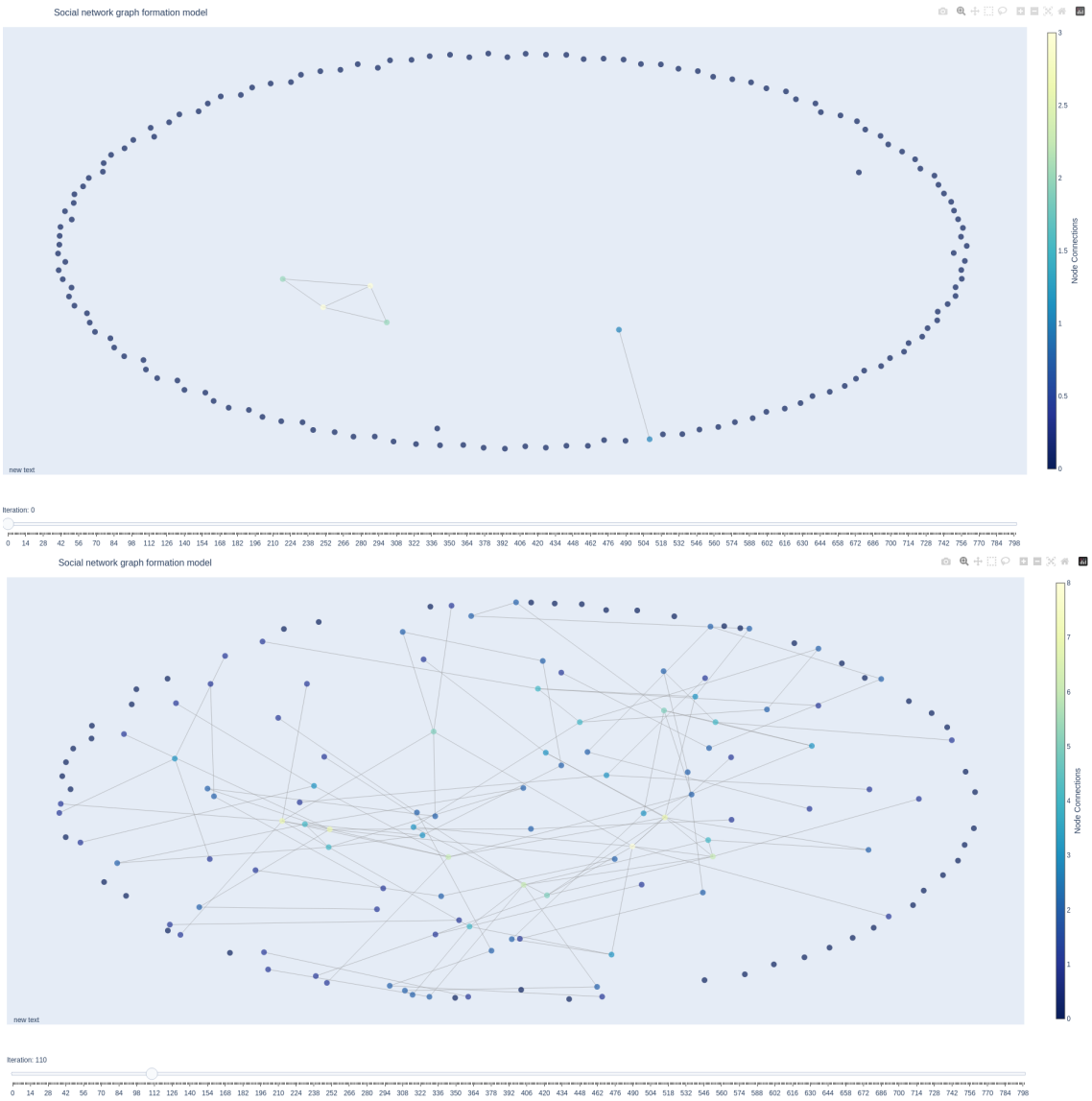
2.5.4 Average Shortest Path Lenght

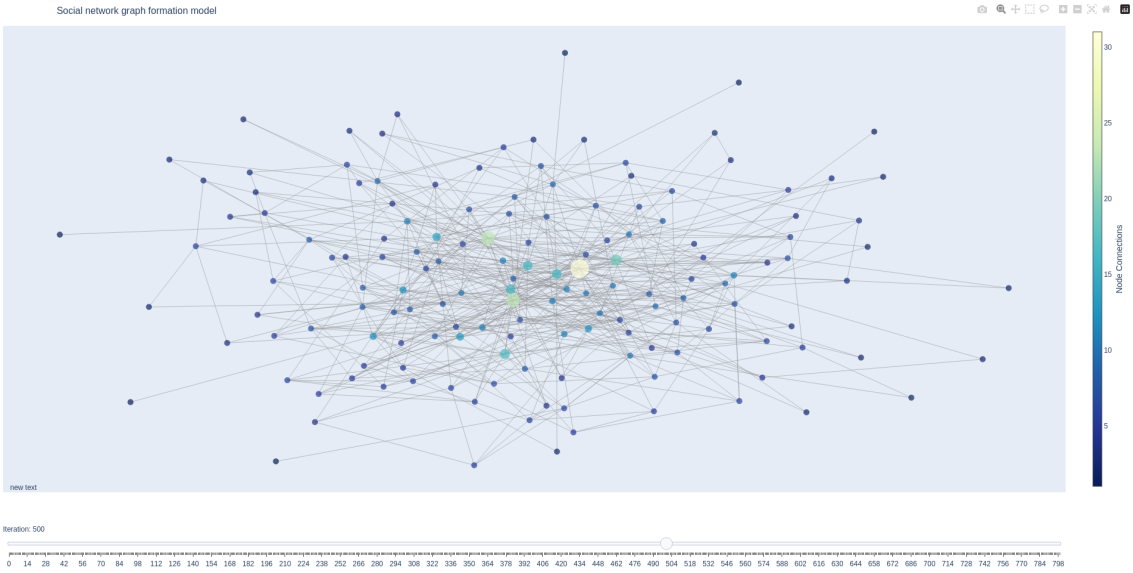
```
def average_shpath_length(G:nx.Graph):
    # Calculate shortest path lengths between all pairs of nodes
    shortest_paths = {}
    for node in G:
        shortest_paths[node] = {}
        for target in G:
            shortest_paths[node][target] = float('inf')
        shortest_paths[node][node] = 0

    for node in G:
        queue = [node]
        while queue:
            current_node = queue.pop(0)
            for neighbor in G[current_node]:
                if shortest_paths[node][neighbor] == float('inf'):
                    shortest_paths[node][neighbor] = shortest_paths[node][current_nod
                    queue.append(neighbor)

    # Compute the sum of all shortest path lengths
    total_path_length = sum(sum(length.values()) for length in shortest_paths.values(

    # Compute the average path length
    num_nodes = len(G)
    avg_path_length = total_path_length / (num_nodes * (num_nodes - 1))

    return avg_path_length
```

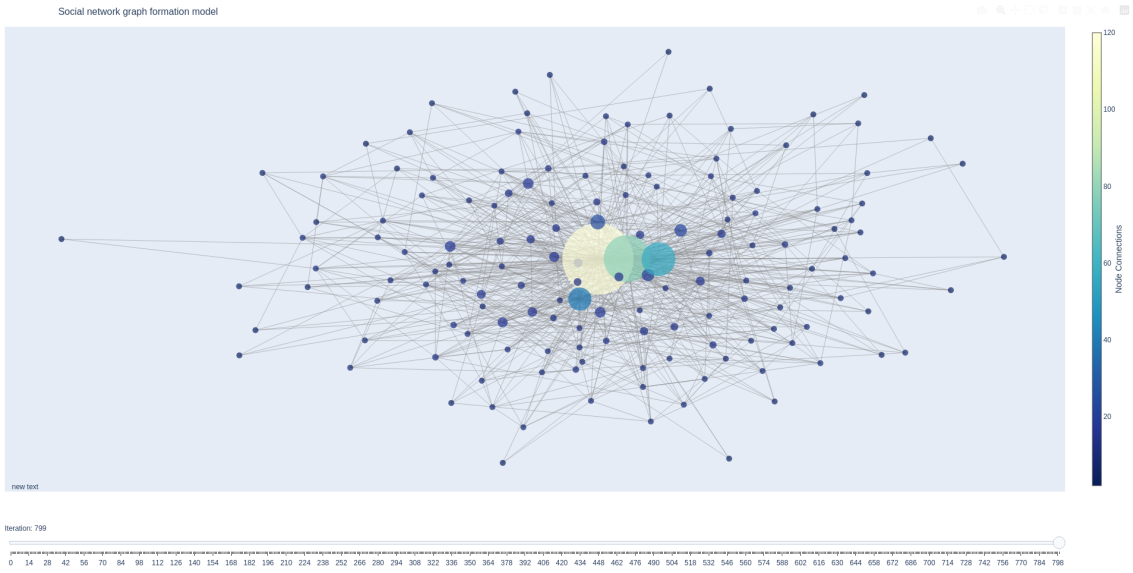
Inizialmente la funzione, come per il **Maximum connected components**, esegue una ricerca in ampiezza a partire da ogni nodo del grafo, aumentando ad ogni iterazione il valore del dizionario `shortest_paths` relativo ad ogni nodo se è la prima volta che viene visitato(per farlo controlla se il valore attuale è ∞).
Dopo la **BFS** la funzione calcola la media delle lunghezze dei cammini trovati.

2.6 Output

L'output del programma è un grafico interattivo in cui possiamo visualizzare uno *snapshot* della rete a varie iterazioni.




Social network graph formation model
new text
Node Connections
30
25
20
15
10
5
Iteration: 500
0 14 28 42 56 70 84 98 112 126 140 154 168 182 196 210 224 238 252 266 280 294 308 322 336 350 364 378 392 406 420 434 448 462 476 490 504 518 532 546 560 574 588 602 616 630 644 658 672 686 700 714 728 742 756 770 784 798


Social network graph formation model
new text
Node Connections
120
100
80
60
40
20
Iteration: 799
0 14 28 42 56 70 84 98 112 126 140 154 168 182 196 210 224 238 252 266 280 294 308 322 336 350 364 378 392 406 420 434 448 462 476 490 504 518 532 546 560 574 588 602 616 630 644 658 672 686 700 714 728 742 756 770 784 798

Le immagini precedenti mostrano l'esempio di un grafo preso a varie epoche, in particolare 1, 110, 500, 800.

Si nota come mano a mano che si avanza con le epoche emergono degli **hub**, che appaiono molto evidenti e con un numero di collegamenti estremamente maggiore degli altri nell'ultima immagine.

Nell'immagine finale si può notare come in questo particolare esempio il grafico presenta un **Giant Component**.

3 Analisi dei Costi

L'analisi dei costi sarà suddivisa in 2 parti, una in cui verranno analizzati i costi relativi alla generazione della rete, e una seconda relativa al costo per il calcolo di ogni metrica.

3.1 Costo Generazione Rete

To compute this cost we firt analyze the single component of the overall computational cost:

- **Costo Preferential Attatchment**
  - *Complessità temporale*: La funzione crea prima un elenco di tutti i nodi nel grafo escluso il nodo dato, che richiede tempo $O(n)$, dove n è il numero di nodi nel grafo.
    Quindi crea tutte le combinazioni del nodo dato con gli altri nodi, il che richiede anche tempo $O(n)$. Quando itera su tutte le combinazioni, calcola il punteggio di *Preferential Attatchment* per ciascun nodo, che richiede tempo $O(1)$ per nodo, quindi tempo $O(n)$ considerando la combinazione con tutti i nodi.
    Infine, seleziona a quale nodo aggiungere un collegamento, pertanto, la complessità temporale complessiva della funzione è $O(n)$.
  - *Complessità spaziale*: La funzione crea un elenco di nodi e un elenco di combinazioni, entrambi occupano $O(n)$ spazio. Memorizza anche uno *score* per ogni nodo nel grafo, che occupa $O(n)$ spazio. Pertanto, la complessità spaziale complessiva della funzione è $O(n)$.

- **Costo Link Prediction**
  - *Complessità temporale*: La funzione inizia con una ricerca in ampiezza dal nodo di partenza, che ha una complessità temporale di $O(n + m)$, con n è il numero di nodi e m numero di archi.
    Successivamente, per ogni nodo a distanza due, calcola l'indice di *Adamic-Adar*, che richiede di iterare su tutti i vicini del nodo, quindi nel caso peggiore potrebbe richiedere un tempo $O(n)$.
    La complessità temporale complessiva della funzione è quindi $O(n(n+m))$.
    Nel calcolo del costo il caso in cui non risulta possibile il calcolo dell'indice di Adamic-Adar non viene tenuto in conto.
  - *Complessità spaziale*:La funzione crea un insieme di nodi a distanza due, che richiede $O(n)$ spazio.
    Inoltre, memorizza uno *score* come per `preferential_attatchment` per ogni nodo nel grafo, che richiede $O(n)$ memoria.
    La complessità spaziale complessiva della funzione è $O(n)$.

Dunque avendo calcolato i singoli costi delle funzioni richiamate per ogni iterazione, risulta evidente che la componente più costosa è l'aggiunta di un link tramite *Adamic-Adar*, quindi la complessità totale nel caso peggiore, in cui tutti i collegamenti sono aggiunti tramite link prediction, per k iterazioni risulta $O(kn(n + m))$, dove k è il numero di iterazioni.

3.2 Costo Calcolo Metriche

Il costo di calcolo delle varie metriche può essere analizzato singolarmente per ogni funzione.

Va notato che per ogni calcolo si suppone che la funzione `degree()` di networkx abbia costo $O(1)$

3.2.1 Diametro

- *Complessità temporale*: La funzione esegue una ricerca in ampiezza da ogni nodo nel grafo, che ha una complessità temporale di $O(n + m)$ per ogni BFS, e dato che viene eseguita per ogni nodo, la complessità temporale complessiva della funzione è $O(n(n + m))$.
- *Complessità spaziale*: La funzione crea un dizionario delle distanze per ogni nodo nel grafo, che richiede $O(n)$ spazio.

Inoltre, utilizza una coda per eseguire la BFS, che nel caso peggiore potrebbe contenere tutti i nodi del grafo, quindi richiede $O(n)$ spazio.
La complessità spaziale complessiva della funzione dunque risulta $O(n)$.

3.2.2 Clustering

- *Complessità temporale*: La funzione itera su ogni nodo nel grafo con complessità di $O(n)$.
  Per ogni nodo, itera su ogni coppia di vicini, con complessità temporale di $O(d^2)$, dove d è il grado del nodo.
  Quindi, la complessità temporale complessiva della funzione è $O(nd^2)$.

- *Complessità spaziale*: La funzione crea un dizionario dei per salvare i valori dei vari coefficienti di clustering per ogni nodo, che richiede $O(n)$ spazio.
  Inoltre, crea una lista dei vicini per ogni nodo, che richiede $O(d)$ spazio per un nodo di grado d.
  Quindi, la complessità spaziale complessiva della funzione è $O(n + d)$.

3.2.3 Maximum Connected Components

- *Complessità temporale*: La funzione itera su ogni nodo nel grafo, con complessità temporale di $O(n)$.
  Per ogni nodo, poi, esegue una ricerca in profondità, con relativa complessità di $O(n + m)$.
  Visto che ogni nodo e ogni arco vengono visitati una sola volta, la complessità temporale complessiva della funzione è $O(n + m)$.

- *Complessità spaziale*: La funzione crea un insieme dei nodi visitati e una lista per il componente connesso in considerazione, entrambi dei quali possono contenere al massimo n nodi, quindi la complessità spaziale è $O(n)$.
  La DFS utilizzata richiede nel caso peggiore $O(n)$ spazio.
  Quindi, la complessità spaziale complessiva della funzione è $O(n)$.

3.2.4 Average Path Length

- *Complessità temporale*: La funzione, come per il **diametro** esegue una ricerca in ampiezza da ogni nodo nel grafo, che ha una complessità temporale di $O(n+m)$ per ogni BFS.
  Dunque il costo in tempo totale risulta $O(n(n + m))$.

- *Complessità spaziale*: La funzione crea un dizionario delle lunghezze dei percorsi più brevi per ogni coppia di nodi nel grafo, che richiede $O(n^2)$ spazio.
  Per la BFS usa una coda di massimo n elementi che ha quidni un costo $O(n)$.
  Quindi, la complessità spaziale complessiva della funzione è $O(n^2)$, dominata dal dizionario.

4 Analisi dei Risultati

Sono stati condotti dei test per analizzare come cambia la rete generata al variare di alcuni parametri.

In particolare è stato scelto 150 come numero di nodi della rete, e poi sono stati cambiati i valori di k e c.
Per k sono stati fatti test con valore di 500, 800 e 1200, mentre per c sono stati fatti 2 test per valori $0 < c \leq 1$ e 2 per $c > 1$, in particolare i valori dei test sono 0.5, 1, 10, 100.

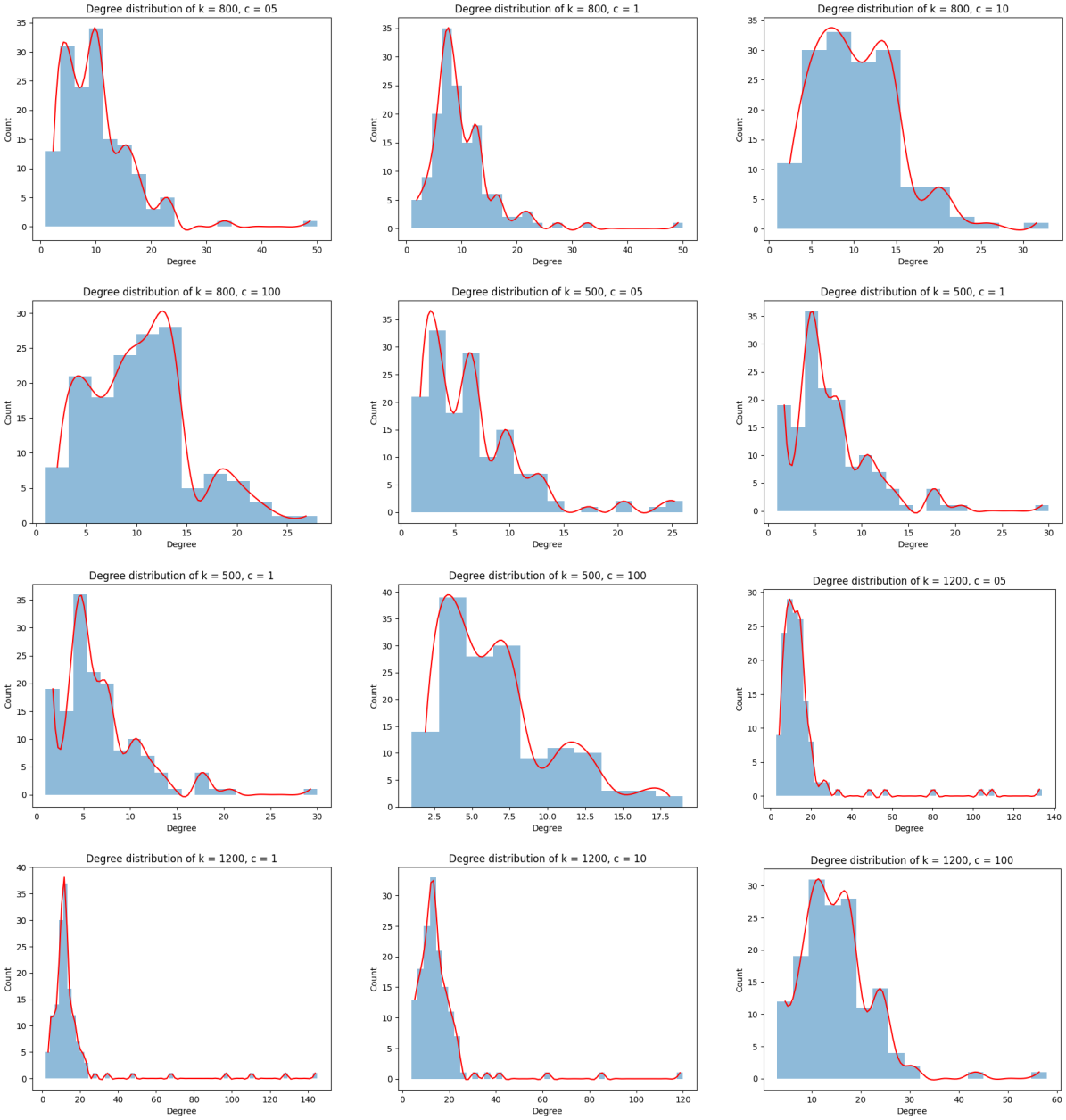
I risultati ottenuti sono i seguenti:

k	c	Diameter	Avg. path	Avg. Clustering coeff.	Average Conncetivity
500	0.5	6	2.847	0.193	4.188
500	1	-	-	0.166	4.264
500	10	6	2.840	0.082	4.361
500	100	-	-	0.073	4.543
800	0.5	5	2.396	0.142	6.960
800	1	5	2.384	0.118	7.136
800	10	5	2.416	0.100	7.319
800	100	4	2.385	0.103	7.517
1200	0.5	3	1.960	0.418	9.704
1200	1	4	1.923	0.493	9.601
1200	10	3	2.010	0.258	10.817
1200	100	4	2.113	0.149	11.183

Dato che in alcune istanze di test il grafo presentava dei nodi di grado 0 il diametro del grafo e l'average path lenght non sono state calcolabili per ogni test, quindi per quei particolari casi di test sono state omesse.

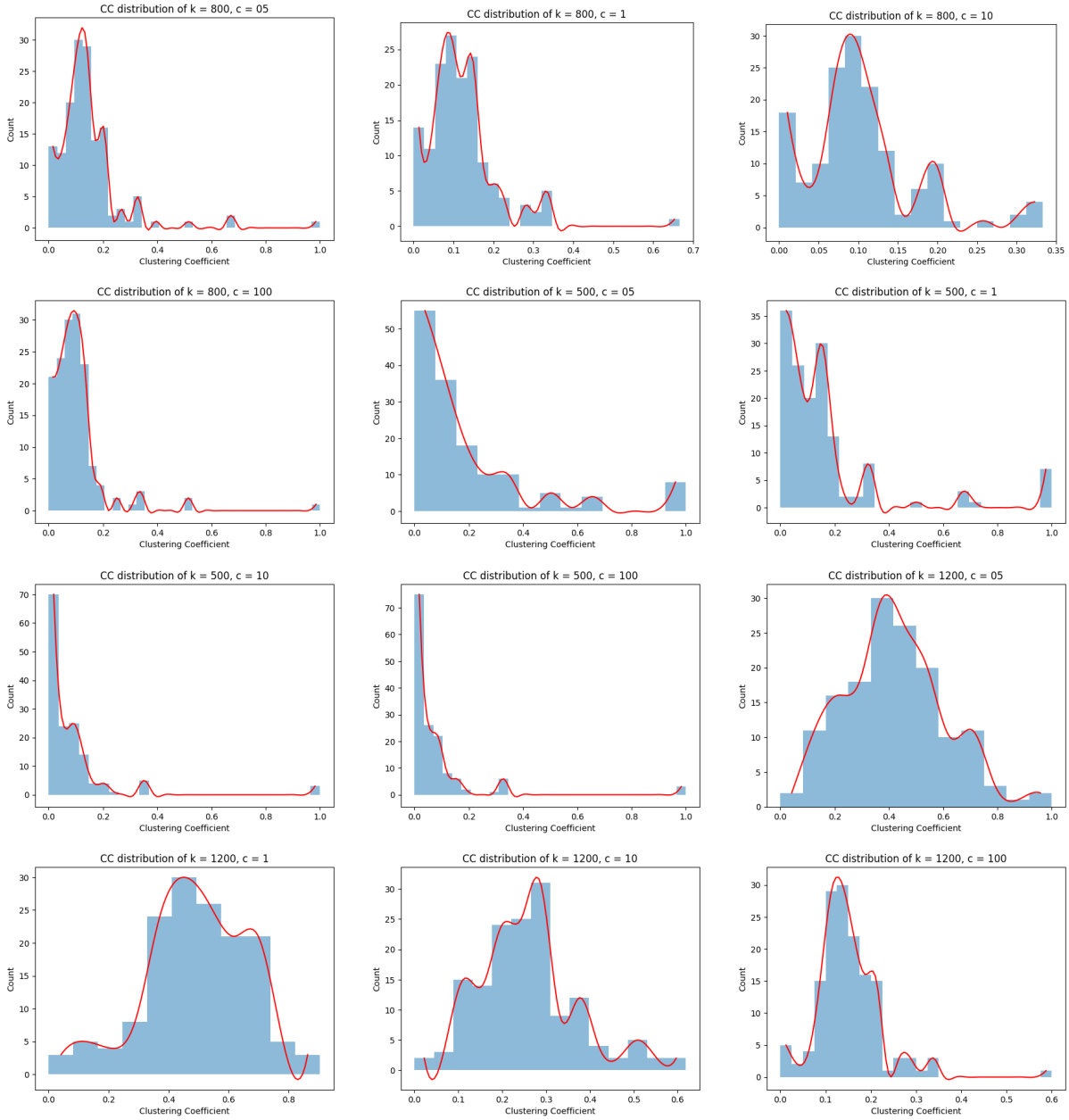
4.1 Nodes degree distribution

Figura 4: Clustering Coefficient Distribution



4.2 Clustering Coefficient Distribution

Figura 5: Degree Distribution



4.3 Osservazioni

Da questi risultati possiamo osservare che:

- all'aumentare di c diminuisce il clustering coefficient, cosa che si rispecchia anche nel grafico della degree distribution, in quanto si vede che è più alta la curva anche per nodi con grado mediamente alto
- se vengono aumentati il numero di iterazioni la distribuione si schiacca a sinistra e presenta **hub** con un numero molto alto di collegamenti, questo si osserva anche nel diametro ,nell'average path lenght, e nel clustering coefficient che fanno comprendere la maggiore connettività del grafo.
- per un numero maggiori di iterazioni i coefficienti di clustering si spostano verso valori intermedi, mentre all'aumentare del valore dei valori di c tendono ad aumentare il numero di nodi con basso clustering, spostando la distribuzione verso sinistra

Riassumendo dunque per c grandi la rete tende ad avere una differenza di degree minore rispetto a quella osservata per $c < 1$, inoltre la distribuzione tende a spostarsi verso le zone centrali per c minori.
Qeusto è un risultato che ci si potrebbe aspettare, dato che un valore di c minore corrisponde una maggiore probabilità di creare collegamenti tramite link prediction, che implementata con Adamic-Addar dà meno importanza a nodi con alto grado per creare nuovi, diversamente dal preferential attatchment, che viene applicato un maggior numero di volte ai nodi quando c è maggiore.

Va inoltre tenuto in considerazione che anche il grado del nodo va a influire sulla decisione del metodo per creare collegamenti, per come è definita p, questo implica che inizialmente, dato che la maggior parte dei nodi ha grado basso si tenderà sempre a preferire il preferential attatchment rispetto al resto per $c > 1$, cosa che andrà mano mano a ridursi con l'aumentare delle iterazioni, e quindi del grado dei vari nodi.

5 Conclusioni

Da questo progetto si può notare come il modello generato sia coerente con una potenziale rete scale free, mostrando capacità di accrescimento che tendono a generare degree distribution simile alla **Power Law** distribution, inoltre a causa dell'utilizzo

del coefficiente di **Adamic-Adar**, si può notare anche che la distribuzione presenta un maggiore numero di nodi con grado medio, rispetto a quanto ci si potrebbe aspettare utilizzando esclusivamente il preferential attatchment.