Università degli Studi di Padova
Dipartimento di Scienze Statistiche
Corso di Laurea Triennale in
Statistica per le Tecnologie e le Scienze

Relazione Finale

Generazione di grafi aleatori con vertici di grado limitato

Relatore Prof. Carlo Ferrari
Dipartimento di Ingegneria dell'Informazione

Laureanda Michela Ropele
Matricola 1222894

Anno Accademico 2022/2023

Indice

Introduzione **1**

1 I grafi e le loro proprietà **3**
- 1.1 Definizioni fondamentali . . . 3
- 1.2 Proprietà dei grafi . . . 6
  - 1.2.1 Connettività . . . 6
  - 1.2.2 Aciclicità . . . 7
  - 1.2.3 Bipartizione . . . 7

2 Programma **9**
- 2.1 Funzione generatrice dei grafi . . . 9
  - 2.1.1 Funzione `choose_nodes_to_adjust()` . . . 9
  - 2.1.2 Funzione `degree_reduction()` . . . 11
  - 2.1.3 Funzione `random_graph_with_hub_nodes()` . . . 13
- 2.2 Funzioni di analisi grafica . . . 19
  - 2.2.1 Funzione `draw_circular_graph()` . . . 19
  - 2.2.2 Funzione `draw_subgraphs()` . . . 19
  - 2.2.3 Funzione `histogram_of_graph_degrees()` . . . 20
- 2.3 Funzioni di confronto . . . 20
  - 2.3.1 Funzione `compare_degree_distributions()` . . . 22
  - 2.3.2 Funzione `plot_3d_property_frequency()` . . . 22
- 2.4 Funzioni di salvataggio e acquisizione da file . . . 23
  - 2.4.1 Funzione `write_graphs_on_file()` . . . 24
  - 2.4.2 Funzione `read_graphs_from_file()` . . . 24
- 2.5 Interfaccia grafica . . . 24

3 Conclusioni **27**
- 3.1 Risultati . . . 27
- 3.2 Possibili sviluppi futuri . . . 30

Appendice **33**

Bibliografia **41**

Introduzione

Negli ultimi anni lo studio dei sistemi di rete ha ricevuto una notevole spinta grazie alla disponibilità sempre crescente di grandi quantità di dati e all'aumento delle capacità di elaborazione informatiche necessarie per la loro conservazione e manipolazione. In particolare, l'ambito di mappatura del World Wide Web è stato uno dei primi ad offrire la possibilità di studiare la topologia dei grafi (o reti complesse) di grandi dimensioni. Gradualmente questo approccio è stato esteso ad altre applicazioni in ambito sociale, economico, biologico e, più in generale, in campo scientifico; infatti, le reti complesse sorgono in un vasto numero di sistemi naturali ed artificiali. Ad esempio, il cervello umano è costituito da miliardi di neuroni interconnessi, gli ecosistemi sono caratterizzati da complesse reti alimentari che descrivono l'interdipendenza tra le specie, i gruppi sociali possono essere rappresentati mediante grafi che illustrano le interazioni tra gli individui, e le grandi infrastrutture di rete, come le reti elettriche e di trasporto, sono fondamentali per la nostra società moderna. Non fa eccezione la cellula vivente, la cui organizzazione e funzione sono il risultato di una complessa rete di interazioni tra geni, proteine ed altre molecole (1).

Nel tempo, i ricercatori hanno iniziato a condurre analisi sistematiche e caratterizzazioni dei grafi per ricercare le leggi che guidano la dinamica e l'evoluzione di questi sistemi complessi. Un risultato centrale di queste attività è che le reti presentano tipicamente topologie complesse e strutture estremamente eterogenee. La teoria dei grafi aleatori, grazie alla sua stretta relazione con la probabilità, offre la possibilità di caratterizzare fenomeni macroscopici in termini di evoluzione dei suoi elementi. Inoltre, l'analisi di grafi che compaiono in campi molto diversi, ha rivelato la presenza di diverse proprietà asintotiche condivise, sollevando la questione della presunta emergenza di studiare reti complesse di carattere più generale (1). L'interesse per le distribuzioni dei gradi, invece, è motivato dal fatto che, nelle reti del mondo reale, le sequenze di gradi, talvolta, mostrano comportamenti notevolmente differenti rispetto a quelli previsti da modelli di grafi aleatori classici, come ad esempio il modello di Erdos-Renyi (5).

Di conseguenza, molti ricercatori hanno provato varie strategie per affrontare questa sfida. Una soluzione ovvia consiste nell'elaborare modelli di grafi casuali che siano in grado di riprodurre la distribuzione dei gradi desiderata permettendo, successivamente, di dedurre ulteriori informazioni attraverso simulazioni o analisi grafiche.

L'idea alla base di questo lavoro è nata dall'interesse sviluppato nei confronti dei grafi come strutture dati complesse per la rappresentazione di dati relazionali e la modellazione di fenomeni che li coinvolgono, nonché dal desiderio di approfondire le conoscenze informatiche di progettazione, implementazione e gestione dei dati.
Il lavoro proposto ha come obbiettivo lo studio e l'implementazione dei grafi aleatori con una particolare attenzione sulla distribuzioni dei gradi dei nodi e la verifica di alcune proprietà di queste strutture. Questo progetto si sviluppa a partire da conoscenze in ambito statistico, probabilistico, informatico e computazionale con particolare concentrazione sulla teoria dei grafi aleatori e la progettazione software. Il risultato finale è un software che consente di simulare grafi aleatori con connessioni di grado vincolate ma casuali tra gli elementi, partendo dal modello più semplice, cioè i grafi aleatori binomiali. Il programma può essere utile per simulare specifici tipi di rete e condurre verifiche generali sulle loro proprietà attraverso la replicazione ed il campionamento ripetuto.

Nel corso della relazione verranno presentate le definizioni di base che saranno utili per la comprensione del lavoro. In particolare, nel Capitolo 1 saranno illustrate le fondamenta teoriche dei grafi e gli strumenti per la loro analisi statistica. Questi strumenti saranno poi impiegati nell'implementazione pratica del software, come descritto nel Capitolo 2. Infine, la relazione si concluderà con una sintesi dei risultati al Capitolo 3 ed un'appendice contenente le funzioni implementate.

Capitolo 1

I grafi e le loro proprietà

Questo primo capitolo introdurrà i concetti principali sulla teoria dei grafi aleatori che saranno poi alla base del modello scelto per l'implementazione del programma. Si presenteranno dapprima le definizioni di grafo indiretto aleatorio e delle sue componenti (vertici, archi e matrice di adiacenza) utili per la generazione della struttura dati. Seguiranno poi la descrizione di densità, grado e nodi centrali al fine di introdurre i grafi aleatori con vertici di grado limitato e, infine, verranno presentate le tre proprietà utilizzate nel prosieguo: connessione, aciclicità e bipartizione.

1.1 Definizioni fondamentali

Un $grafo$ è una struttura dati complessa utile per la rappresentazione di informazioni relazionali ed è costituito da due unità fondamentali: gli elementi detti *vertici* (o *nodi*) e delle coppie di nodi che rappresentano l'esistenza di una relazione (nel caso presente binaria) tra due elementi, dette *archi*.

Definizione 1.1 Un **grafo indiretto** (o *non orientato*) G è definito da una coppia ordinata di insiemi $G = (V, E)$ dove $V = \{1, 2, ..., n\}$ è un insieme non vuoto di elementi numerabili chiamati *nodi* (o vertici), mentre $E = \{(i, j) : i, j \in V, i \neq j\}$ è un insieme non ordinato di coppie di nodi chiamate *archi* tali che $(i, j) = (j, i)$ (1).

L'arco (i, j) unisce i vertici i e j, che si dicono *adiacenti* o *connessi*. Il numero totale di vertici $n = |V|$ definisce l'*ordine* del grafo. I grafi non orientati sono rappresentati da un insieme di punti etichettati (i vertici) uniti a coppie da linee (gli archi corrispondenti).

Definizione 1.2 Un **grafo aleatorio (binomiale)** $G(n, p)$ di ordine n con probabilità di connessione p è un grafo indiretto con n nodi dove ciascuna coppia di vertici (i, j) è connessa in modo indipendente con una probabilità p costante tale che $0 < p < 1$ (4).

La loro definizione si basa sulla generazione casuale dei collegamenti tra i nodi attraverso una distribuzione probabilistica che ne determina la probabilità di connessione. La creazione di una connessione tra due elementi è un evento casuale indipendente dalle caratteristiche dei vertici che rende l'esistenza di ciascun arco equiprobabile rispetto a tutti gli altri. In questa classe di grafi si assume un'assoluta mancanza di conoscenza dei principi che guidano la creazione degli archi, rendendo difficile osservare sottogruppi di nodi per via del principio di non organizzazione che guida la formazione delle connessioni (1). L'immagine in Figura 1.1 mostra un esempio di grafo aleatorio indiretto.

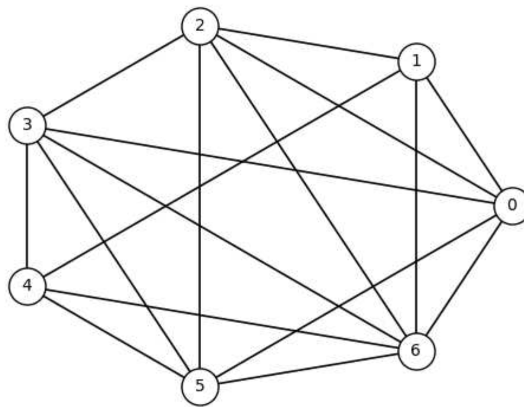

FIGURA 1.1: Rappresentazione circolare di un grafo aleatorio indiretto di ordine 7 con probabilità di connessione $p = 0.8$.

Definizione 1.3 Una **matrice di adiacenza** $\mathbf{X} = (x_{ij})$ è una matrice binaria quadrata di ordine n tale che $\forall i, j \in V$

$$x_{ij} = \begin{cases} 1 & \text{se } (i,j) \in E \\ 0 & \text{se } (i,j) \notin E \end{cases} \tag{1.1}$$

Per i grafi indiretti la matrice di adiacenza è simmetrica poiché $x_{ij} = x_{ji}$ (1).

La generazione e l'implementazione successiva dei grafi avverrà attraverso la matrice di adiacenza, e sottoforma di array bidimensionale, per via della sua chiarezza concettuale e della facilità di visualizzazione, nonché per l'accesso veloce alle informazioni sulla connettività dei nodi. Questo metodo faciliterà il calcolo di attributi come il grado dei nodi.

Definzione 1.4 La **densità** di un grafo indiretto $G = (V, E)$ di ordine n è definita come

$$D = \frac{|E|}{\binom{n}{2}} = \frac{2|E|}{n(n-1)} \quad (2) \tag{1.2}$$

A parole, la densità è il rapporto tra il numero di archi osservati ed il numero massimo di archi possibili $\binom{n}{2}$. Un grafo con $D \to 0$ è detto *sparso* mentre un grafo con $D \to 1$ è detto *denso*. Se $|E| = \binom{n}{2}$, ovvero $D = 1$ il grafo è detto *completo* perché tutte le possibili coppie di vertici sono connesse (1). In seguito, verranno studiate le proprietà dei grafi, le quali saranno garantite o meno anche in funzione della densità del grafo.

Definizione 1.5 Il **grado** d_i del vertice i è definito come il numero di archi del grafo incidenti sul vertice i

$$d_i = \sum_j x_{ij} \tag{1.3}$$

La *distribuzione dei gradi P(d)* di un grafo indiretto è definita come la probabilità che un vertice scelto a caso abbia grado d e si ottiene costruendo l'istogramma normalizzato del grado dei nodi di un grafo (1). Il *grado medio* $\overline{d}$, invece, è definito come il valore medio di d calcolato su tutti i nodi.

Definizione 1.6 I vertici con un grado elevato, ovvero con un grande numero di connessioni rispetto alla media, sono detti **nodi centrali**.

Questo tipo di nodi è molto rilevante in quanto mette in relazione un numero elevato di vertici ma, in un grafo aleatorio con distribuzione uniforme degli archi, questo tipo di nodi è difficile da osservare poiché ciascun arco ha la medesima possibilità di essere generato. Affinché un grafo aleatorio, generato a partire da un algoritmo di implementazione randomica, presenti una parte di nodi con un numero di collegamenti superiore alla media, è necessario imporre dei vincoli sui valori della matrice di adiacenza.

In questo lavoro si indicherà come *grafo aleatorio con vertici di grado limitato* qualsiasi grafo aleatorio generato imponendo dei vincoli sul grado dei vertici. La Figura 1.2 mostra un esempio in cui ciascun nodo ha grado inferiore o uguale a 5.

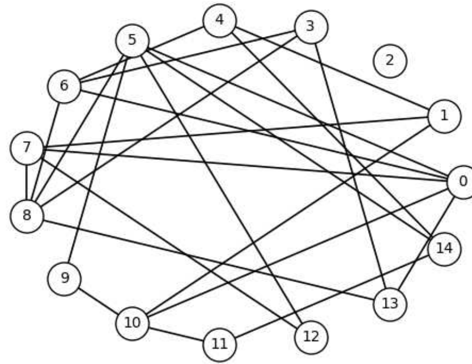

FIGURA 1.2: Grafo aleatorio di ordine 15 e con probabilità di connessione $p = 0.3$ e grado massimo pari a 5.

Nel presente caso di studio, l'algoritmo di generazione implementato prevede come parametri in ingresso il numero di nodi, la probabilità di esistenza degli archi, un parametro per inserire la proporzione di nodi centrali, ovvero la proporzione di vertici senza vincoli e con grado arbitrario, ed un parametro per indicare il grado massimo dei nodi rimanenti.

1.2 Proprietà dei grafi

Sia $G = (V, E)$ un grafo indiretto e sia $< v_1, ..., v_k >$ una sequenza ordinata di vertici in V in cui ogni coppia consecutiva di nodi forma un arco. Se $v_1 \neq v_k$, ovvero non è presente un arco che congiunge il nodo iniziale v_1 al nodo finale v_k, la sequenza è detta *cammino* ed ha lunghezza pari al numero di archi che lo compongono, mentre se $v_1 = v_k$ la sequenza è detta *ciclo*. Si osservi che se $k = 2$ il cammino considerato è formato da un singolo arco, e che tutti gli archi in E individuano un cammino di lunghezza unitaria tra i nodi coinvolti. Diremo, inoltre, che il vertice v_i è *raggiungibile* da vertice v_j se esiste almeno un cammino che congiunge il nodo v_i al nodo v_j.

1.2.1 Connettività

G è definito **connesso** se per ogni coppia di vertici $(i, j) : i, j \in V$ esiste almeno un cammino che collega il nodo i al nodo j. Nei grafi non orientati, affinché questo avvenga, è condizione necessaria la non esistenza di componenti isolate.

L'immagine in Figura 1.3 è un esempio di grafo connesso poiché ciascun vertice è raggiungibile partendo da un qualsiasi altro vertice.

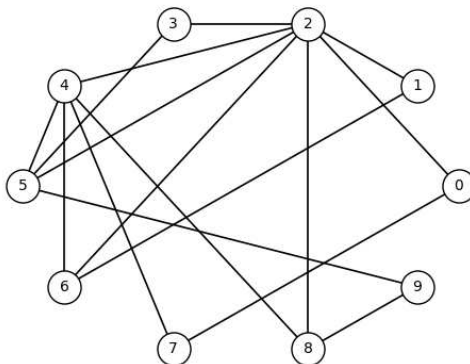

FIGURA 1.3: Rappresentazione circolare di un grafo aleatorio indiretto connesso di ordine 10 e con probabilità di connessione $p = 0.3$.

1.2.2 Aciclicità

Un grafo **aciclico** è un grafo senza *cicli*, ovvero scelto uno qualsiasi tra i nodi del grafo non è possibile partire da esso e tornare ad esso percorrendo ciascun cammino presente all'interno del grafo diverso da quello di partenza. Graficamente questa proprietà si presenta come assenza di gruppi di nodi con formazione chiusa ad anello.

L'immagine in figura 1.4(a) rappresenta un grafo aciclico perché ogni cammino possibile si interrompe prima di tornare al nodo di partenza. L'immagine 1.4(b), invece, rappresenta un grafo ciclico perché è possibile partire dal nodo 3 e raggiungere il nodo 1 attraverso il cammino (3, 6, 1), tornando poi al nodo 3 attraverso il cammino (diverso) (1, 0, 4, 3).

1.2.3 Bipartizione

Un grafo **bipartito** è definito tale se l'insieme dei vertici è partizionabile in due sottoinsiemi separati tali che ogni vertice di ciascun sottoinsieme è collegato solo a vertici dell'insieme opposto.

L'immagine in Figura 1.5 rappresenta un grafo bipartito in cui i due sottoinsiemi sono rispettivamente $\{0, 1, 2\}$ e $\{3, 4, 5, 6, 7\}$.

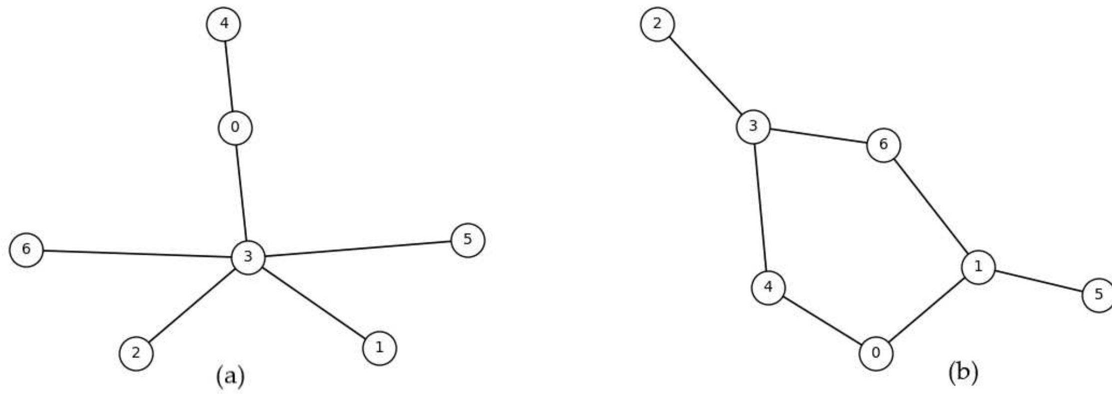

Figura 1.4: Rappresentazione di un grafo aleatorio indiretto aciclico di ordine 7 con probabilità di connessione $p = 0,2$ (a) e rappresentazione di un grafo aleatorio indiretto non aciclico di ordine 7 con probabilità di connessione $p = 0,2$ (b).

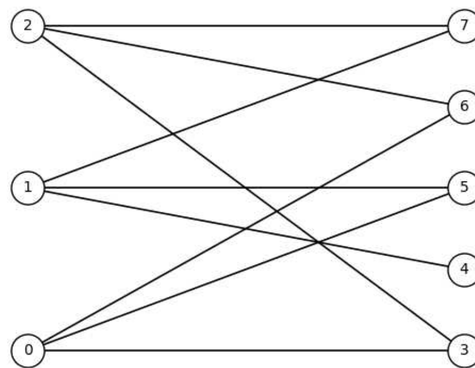

Figura 1.5: Rappresentazione multipartita di un grafo aleatorio indiretto bipartito di ordine 7 con probabilità di connessione $p = 0,2$.

Capitolo 2

Programma

Questo capitolo ha lo scopo di presentare ed analizzare il programma scritto in Python (versione 3.10.4), spiegando dettagliatamente le istruzioni delle funzioni coinvolte nella generazione dei grafi con vertici di grado limitato. Seguirà una sezione dedicata alle funzioni di rappresentazione grafica ed una per le funzioni dedicate al salvataggio e alla lettura dei grafi generati. Questo software è reso facilmente accessibile grazie all'implementazione di un'interfaccia grafica per semplificare l'interazione con l'utente.
Va ricordato che il modello implementato parte dall'ipotesi di fissare un numero intero positivo di nodi $n > 1$, una probabilità uniforme $p \in (0, 1)$, che denota la probabilità dell'esistenza di un arco tra ciascuna coppia di nodi all'interno del grafo, e dei parametri opzionali relativi all'ordine massimo dei nodi: d_{max} per indicare il grado massimo dei nodi non centrali e π per indicare la proporzione dei nodi centrali, se presenti.

2.1 Funzione generatrice dei grafi

Qui di seguito vengono esaminate dapprima le due funzioni `choose_nodes_to_adjust()` e `degree_reduction()` fondamentali nel processo di controllo e modifica del grado dei nodi e, successivamente, la funzione `random_graph_with_hub_nodes()` di generazione vera e propria dei grafi.

2.1.1 Funzione `choose_nodes_to_adjust()`

La funzione `choose_nodes_to_adjust()` serve a selezionare, tra tutti i nodi con un grado superiore al vincolo imposto da d_{max}, i nodi centrali, ovvero quelli che manterranno un numero di connessioni superiore al vincolo. Per effettuare questa selezione, la funzione riceve in input:

- il numero totale di nodi n salvato in `n`;
- un vettore contenente i gradi dei nodi d_i $\forall i \in V$ salvati in `d`;
- la proporzione di nodi centrali π salvata in `ratio_hub_nodes`;
- il grado massimo d_{max} consentito a tutti i nodi diversi da quelli centrali, salvato in `max_degree`.

Il risultato ottenuto dalla funzione è una lista di nodi che richiedono una limitazione del grado per rispettare i vincoli imposti.

In primo luogo, viene calcolato il numero di nodi centrali sui nodi totali in base al parametro di proporzione π specificato. Successivamente viene inizializzata la lista `is_hub_nodes` che conterrà tutti i nodi di grado superiore al limite imposto da `max_degree`. Il ciclo `for()` a riga 5 scorre la lista di gradi del grafo e salva in `is_hub_nodes` gli indici dei nodi con grado superiore al massimo. Se la lunghezza di questa lista non supera il numero di nodi centrali possibili, allora i vincoli sono rispettati ed il grafo non necessita di una correzione, facendo restituire alla funzione una lista vuota. Se invece ciò non si verifica, si procede con la selezione casuale dei nodi centrali e la restituzione della lista contenente i nodi che necessitano della correzione dei loro gradi per rispettare i vincoli imposti.

```python
1  def choose_nodes_to_adjust(n, d, ratio_hub_nodes, max_degree):
2      n_hub_nodes = int(n*ratio_hub_nodes) #numero di nodi con grado
   arbitrario
3      is_hub_node = [] #lista di nodi con grado superiore al massimo
4
5      for i in range(n):
6          if d[i] > max_degree:
7              is_hub_node.append(i)
8
9      if len(is_hub_node) <= n_hub_nodes:
10         nodes_to_adjust = [] #i nodi rispettano le specifiche: non
   serve abbassare i gradi
11
12     else: #se ci sono troppi nodi centrali si procede con la selezione
13         choose_hub_nodes = np.random.choice(is_hub_node, n_hub_nodes,
   replace=False)
14         nodes_to_adjust = np.setdiff1d(is_hub_node, choose_hub_nodes)
15
16     return nodes_to_adjust
```

Listato 2.1: Funzione di scelta dei nodi con imposizione del vincolo.

2.1.2 Funzione `degree_reduction()`

La funzione `degree_reduction()` ha lo scopo di effettuare la riduzione dei gradi dei nodi di un grafo in modo che quest'ultimo rispetti i vincoli imposti. Per farlo sono richiesti in input i seguenti parametri:

- la matrice di adiacenza salvata nell'oggetto `m`;
- il grafo da modificare salvato nell'oggetto `g`;
- il grado massimo richiesto d_{max} salvato in `max_degree`;
- la lista degli indici che identificano i nodi di cui ridurre il grado salvata in `index`.

Al termine della procedura, la funzione restituirà il grafo e la matrice di adiacenza aggiornati garantendo che i nodi presenti all'inizio in `index` rispettino il vincolo di grado imposto.

La sequenza di istruzioni inizia calcolando i gradi dei nodi in `d` attraverso la somma per righe dei valori della matrice di adiacenza. Successivamente, per ogni nodo i appartenente alla lista `index`, la funzione inizializza una lista vuota `old_edges` e calcola il numero esatto di archi il cui valore dovrà essere posto a 0. A questo punto, dopo aver aggiornato con il ciclo `for()` a riga 9 la lista `old_edges` con tutti gli indici dei nodi connessi ad i, si procede con la selezione randomica degli archi da spegnere ed il successivo azzeramento degli elementi (i, j) e (j, i) corrispondenti nella matrice di adiacenza (poiché, ricordo, quest'ultima è simmetrica), seguito dalla rimozione del rispettivo arco all'interno grafo. Poiché la libreria `NetworkX` memorizza un grafo non orientato salvando ciascun arco (i, j) con l'indice di valore maggiore in prima posizione, seguito dall'indice di valore inferiore, il ciclo `if()` presente a riga 20 garantisce la corretta rimozione degli archi tenendo conto di questo fatto.

```
def degree_reduction(m, g, max_degree, index):
    d = m.sum(axis = 1)
    n = len(d)
    for i in index: #scorrimento delle righe della matrice di adiacenza con grado > max_degree
        while d[i] > max_degree:
            old_edges = [] #lista di archi accesi
            to_be_off = int(d[i] - max_degree) #numero di archi da spegnere

            for j in range(n):   #scorrimento delle colonne
                if m[i][j] == 1:
                    old_edges.append(j)  #salvataggio dell'elemento  j-esimo nella lista

            off = np.random.choice(old_edges,to_be_off, replace=False)
            #estrazione casuale degli archi da spegnere tra gli archi accesi

            for k in off: #spegnimento dei nodi selezionati
                m[i][k] = 0
                m[k][i] = 0 #perche' la matrice e' simmetrica

                if(i < k):
                    g.remove_edge(i, k)
                else:
                    g.remove_edge(k, i)

    return (g, m)
```

Listato 2.2: Funzione di correzione del grado dei nodi.

2.1.3 Funzione `random_graph_with_hub_nodes()`

La funzione `random_graph_with_hub_nodes()` permette di generare dei grafi aleatori di ordine n con probabilità di connessione pari a p ed, eventualmente, un grado massimo d_{max} ad eccezione di una proporzione di nodi centrali π. Questa funzione riceve in input:

- il numero di nodi n salvato in `n`;
- la probabilità di connessione p salvata in `p`;
- il grado massimo d_{max} salvato in `max_degree` (`None` di default);
- la proporzione di nodi centrali π salvata in `ratio_hub_nodes` (0 di default);
- la possibilità di stampare i dettagli del grafo con `print_details` (`False` di default).

Al termine la funzione restituirà un oggetto di tipo `Graph` contenente un grafo aleatorio che rispetta le caratteristiche specificate.

Le prime righe della procedura (3-23) servono ad eseguire dei controlli sui parametri in input per verificare che questi rispettino i vincoli di esistenza. Se ciò dovesse non avvenire la funzione stampa un messaggio di errore interrompendo l'esecuzione del programma. In particolare, deve valere che n sia intero positivo, p un valore in virgola mobile nell'intervallo $(0, 1)$, così come la proporzione di nodi centrali π, e il grado massimo d_{max} un intero positivo minore del numero di nodi (al più $n - 1$). Inoltre, viene aggiunto il controllo per verificare che in presenza di nodi centrali sia indicato anche il grado massimo; in caso contrario la funzione non avrebbe sufficienti informazioni sulla selezione dei nodi centrali e non potrebbe proseguire con la loro scelta.
Per la generazione del grafo vero e proprio si procede con la selezione degli archi attraverso un processo randomico con probabilità uniforme. Questo modello assume la dipendenza tra gli archi e le caratteristiche dei nodi, portando ad osservare delle connessioni distribuite perlopiù uniformemente all'interno del grafo e, quindi, analogamente anche all'interno della matrice di adiacenza.

La scelta implementativa sottostante calcola il numero di archi del grafo completo e poi li seleziona uniformemente con probabilità p. Questo valore viene dal fatto che ciascuno degli n nodi può avere al più $n-1$ connessioni (uno verso ogni altro nodo escluso sé stesso) ma, poiché il grafo è indiretto (quindi non si contano gli archi duplicati e, in questo caso, si esclude l'elemento diagonale), il numero di archi da calcolare è l'intero

N ottenuto attraverso la formula combinatoria

$$N = \binom{n}{2} = \frac{n(n-1)}{2} \tag{2.1}$$

Per eseguire la selezione si genera un vettore `edges` contenente l'informazione sulla presenza o l'assenza di ciascuno tra i possibili archi. Data l'indipendenza tra archi e le caratteristiche dei nodi, si utilizza un'istruzione di generazione randomica di valori Bernoulliani distribuiti come una variabile aleatoria binomiale di parametri $n = 1$ e p, ovvero

$$X_{ij} \sim Bi(1, p) \tag{2.2}$$

con $i \neq j \ \forall \ i, j \in V$.

Ottenute le realizzazioni delle variabili indicatrici che determinano l'esistenza o meno degli archi di peso unitario costante, si procede, dopo aver inizializzato il grafo e la matrice di adiacenza a valori nulli, popolando prima la lista dei nodi, con l'aggiunta dei valori da 0 a $n-1$, e poi la matrice di adiacenza assieme alla lista degli archi. Vengono utilizzati gli indici *index*, i, j per scorrere, rispettivamente, il vettore di archi randomici, le righe e le colonne della matrice. I cicli for annidati nelle righe 37-48 scorrono contemporaneamente la lista di archi ed il triangolo superiore della matrice garantendo che ciascuna coppia di indici venga esaminata una singola volta e si eviti di considerare archi duplicati: il ciclo esterno scorre tutti i nodi da 0 a $n-1$ determinando il primo nodo dell'arco potenziale i, mentre il ciclo interno scorre tutti i nodi successivi ad i, da $i+1$ a $n-1$, e gli elementi della lista degli archi. Questo tipo di avanzamento riduce al minimo il numero di archi generati. Quindi, se per ogni ciclo interno l'elemento *index* di `edges` è uguale a 1, significa che esso indica la presenza dell'arco (i, j) portando così ad aggiornare l'elemento della matrice di riga i e colonna j con un valore booleano positivo e con la successiva aggiunta al grafo dell'arco corrispondente. Se il valore in *index* è uguale a 0, l'arco non esiste e lo scorrimento prosegue senza eseguire azioni. Al termine di ogni confronto sull'elemento di posto (i, j) l'indice *index* viene incrementato di 1. A questo punto è necessario aggiornare il triangolo inferiore della matrice ma, poiché essa deve essere simmetrica, per farlo è sufficiente sommare sé stessa con la sua trasposta. Segue infine una conversione dei valori della matrice in interi necessaria per evitare errori nel codice futuro.

Le righe 54-71 si occupano della limitazione di gradi nel caso in cui siano stati specificati i vincoli di grado. La prima istruzione all'interno del ciclo `if()` si occupa di calcolare, prima delle eventuali modifiche, il vettore `before` di gradi iniziali dei singoli

nodi, effettuando la somma per righe della matrice di adiacenza. Successivamente vengono inizializzati un vettore vuoto `pawns`, che conterrà gli indici dei gradi da modificare, ed `after`, posto uguale a `before`, che conterrà i gradi dei nodi finali oppure resterà invariato nel caso in cui non sia necessario effettuare alcuna operazione di modifica. A questo punto viene verificata la presenza o meno di gradi centrali: in caso affermativo viene chiamata la funzione `choose_nodes_to_adjust()` per la verifica che i nodi con grado superiore al massimo siano della giusta percentuale. Se ciò non accade, e si verifica che il vettore di nodi con grado da aggiustare `pawns` ha lunghezza diversa da zero, significa che è necessario intervenire sul grado di alcuni di essi per apportarne la riduzione. Questi nodi sono selezionati a riga 59-60 e modificati a riga 69. In caso di assenza di nodi centrali deve essere corretto ciascun nodo con grado superiore al massimo e ciò viene eseguito da con un ciclo `for()` (righe 63-66). La funzione `degree_reduction()` esegue l'operazione di riduzione e restituisce il nuovo grafo assieme alla nuova matrice di adiacenza che, a questo punto, rispettano i vincoli assegnati.

L'ultima sezione della funzione è dedicata alla stampa delle caratteristiche del grafo tra cui il vettore dei gradi dei nodi (nel caso di limitazione dei gradi prima e dopo le modifiche con l'informazione sugli archi spenti), il numero di archi osservati sul numero di archi possibili, la densità del grafo, la matrice di adiacenza, la lista de nodi e la lista degli archi.

```
def random_graph_with_hub_nodes(n, p, max_degree = None,
    ratio_hub_nodes = 0, print_details = False):

    #controllo dei parametri in input:
    if(p < 0 or p > 1):
        print("Error: p must be in [0,1].\n")
        return None

    if(n < 0 or (n % 1 != 0)):
        print("Error: n must be a positive integer.\n")
        return None

    if(ratio_hub_nodes < 0 or ratio_hub_nodes > 1):
        print("Error: ratio_popular_nodes must be in [0,1].\n")
        return None

    if(ratio_hub_nodes != 0 and max_degree == None):
        print("Error: with ratio_popular_nodes choose max_degree.\n")
        return None

    if max_degree is not None:
        if(max_degree < 0 or max_degree > n or (max_degree % 1 != 0)):
            print("Error: max_degree must be positive integer in [0, n
    ].\n")
            return None

    #creazione del grafo e della matrice di adiacenza:
    n_values = int((n**2-n) / 2) #numero di archi possibili
    edges = [np.random.binomial(1, p) for i in range(n_values)] #
    selezione randomica degli archi
    adjacency_matrix = np.zeros((n,n)) #inizializzazione della matrice
     di adiacenza

    Graph = nx.Graph() #grafo vuoto

    for i in range(n):
        Graph.add_node(i) #aggiunta dei nodi

    #aggiunta degli archi:
    index = 0
    for i in range(n):                      #indice di riga
        for j in range(i+1, n):             #indice di colonna
            if edges[index] == 1:           #se esiste l'arco
```

```
40                  adjacency_matrix[i][j] = 1  #aggiornamento del
    triangolo superiore della matrice di adiacenza
41
42                  #aggiunta archi
43                  if i<j:
44                      Graph.add_edge(i, j)
45                  else:
46                      Graph.add_edge(j, i)
47
48              index += 1
49
50      adjacency_matrix += adjacency_matrix.T     #aggiornamento del
    triangolo inferiore
51      adjacency_matrix = adjacency_matrix.astype(int)
52
53      #operazioni sui vincoli di grado:
54      if max_degree is not None: #in caso di presenza di un grado
    massimo
55          before = adjacency_matrix.sum(axis=1) #gradi dei nodi di
    partenza
56          pawns = [] #lista di nodi con grado da limitare
57          after = before #se si rispettano le specifiche
58
59          if( 0 < ratio_hub_nodes < 1):
60              pawns = choose_nodes_to_adjust(n, before, ratio_hub_nodes,
     max_degree)
61
62          #aggiustamento dei nodi con grado superiore a max_degree:
63          elif ratio_hub_nodes == 0:
64              for i in range(n):
65                  if(before[i] > max_degree):
66                      pawns.append(i)
67
68          if len(pawns) != 0: #riduzione dei gradi
69              z = degree_reduction(adjacency_matrix, Graph, max_degree,
    pawns)
70              Graph = z[0] #nuovo grafo
71              adjacency_matrix = z[1] #nuova matrice
72              after = adjacency_matrix.sum(axis=1) #gradi finali
    conformi alle specifiche
73
74      #stampa dei dettagli:
75      if print_details == True:
76
```

```
77        n_edges = nx.number_of_edges(Graph)
78        density = round(nx.density(Graph), 3)
79
80        if max_degree != None:
81            print("\nGradi prima:", before, "\n",
82                  "\nGradi dopo: ", after, "\n",
83                  "\nArchi spenti =", int(sum(before - after)/2),"\n")
84        else:
85            print("\nGradi: ", adjacency_matrix.sum(axis=1), "\n")
86
87        print("Numero di archi:", n_edges,
88              "su", n*(n-1), "possibili.\n\nDensita' del grafo:",
    density,
89              "\n\nMatrice di adiacenza:\n", adjacency_matrix,
90              "\n\nNodi:", list(Graph.nodes))
91        print("\nArchi:")
92        pprint.pprint(list(Graph.edges))
93    return Graph
```

Listato 2.3: Funzione generatrice di grafi aleatori con vertici di grado limitato.

2.2 Funzioni di analisi grafica

In questo paragrafo viene presentata una lista di funzioni di rappresentazione dei grafi utili a mettere in luce diversi aspetti delle strutture dati generate. Ciascuna delle tre funzioni riceve in input il grafo e ne restituisce il grafico. Il codice citato sarà consultabile in Appendice: Listato 2.

2.2.1 Funzione `draw_circular_graph()`

La prima funzione grafica fornisce una rappresentazione del grafo ordinata e pulita attraverso la disposizione in cerchio dei nodi, facilitando l'individuazione dei nodi centrali ed il tracciamento dei percorsi da un nodo ad un altro. In questo modo si agevola particolarmente la visualizzazione di grafi con un numero di nodi inferiore o uguale a 25 e con una densità non particolarmente elevata.
La Figura 2.1 mostra un esempio di grafo aleatorio con vertici di grado limitato in cui si può osservare la presenza dei tre nodi centrali 1, 8 e 13 con, rispettivamente, 5, 5 e 6 connessioni ciascuno.

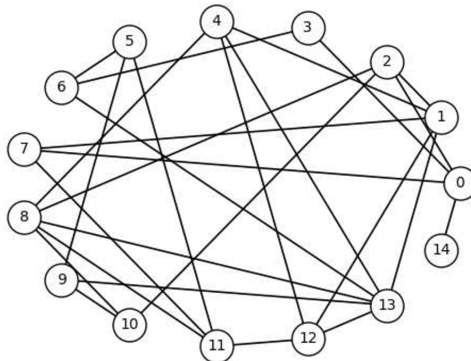

Figura 2.1: Rappresentazione circolare di un grafo aleatorio indiretto di parametri $n = 15$, $p = 0.3$, grado massimo pari a 4 e proporzione di nodi centrali pari a 0.2.

2.2.2 Funzione `draw_subgraphs()`

La seconda funzione grafica rappresenta i grafi mettendo in evidenza l'esistenza di eventuali sottogruppi e colora ciascun sottogruppo con una diversa tonalità di azzurro. Questa funzione offre una buona rappresentazione per grafi fino ad un numero di nodi

inferiore al centinaio.
La Figura 2.2 mostra un esempio di grafo aleatorio diviso in due sottografi.

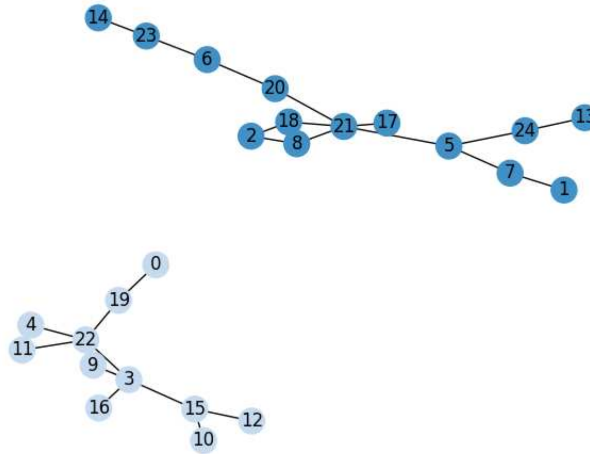

FIGURA 2.2: Rappresentazione di un grafo aleatorio indiretto di parametri $n = 25$, $p = 0.3$, grado massimo pari a 2 e proporzione di nodi centrali pari a 0.1 diviso in due sottografi.

2.2.3 Funzione `histogram_of_graph_degrees()`

La terza funzione grafica genera un istogramma di frequenza per la distribuzione dei gradi dei nodi. Sull'asse delle ascisse sono presenti i gradi dei nodi d_i con $i \in V$, mentre sull'asse delle ordinate la proporzione dei nodi con grado d_i.
Le figure che seguono mostrano tre esempi di istogramma dei gradi per, rispettivamente, un grafo aleatorio semplice (Figura 2.3), un grafo aleatorio con limitazione dei gradi (Figura 2.4) e un grafo aleatorio con limitazione dei gradi ad eccezione di una proporzione di nodi centrali (Figura 2.5).

2.3 Funzioni di confronto

Le seguenti funzioni permettono di mettere a confronto due o più elementi al variare dei parametri: la prima paragona la distribuzione dei gradi in grafi con diversi valori di p, mentre la seconda calcola la proporzione di grafi in campioni aventi una determinata proprietà al variare del numero di nodi n e della probabilità di connessione p. Entrambe le funzioni servono ad osservare la variazione delle caratteristiche del grafo applicando vincoli differenti. Il codice citato sarà consultabile in Appendice: Listato 3.

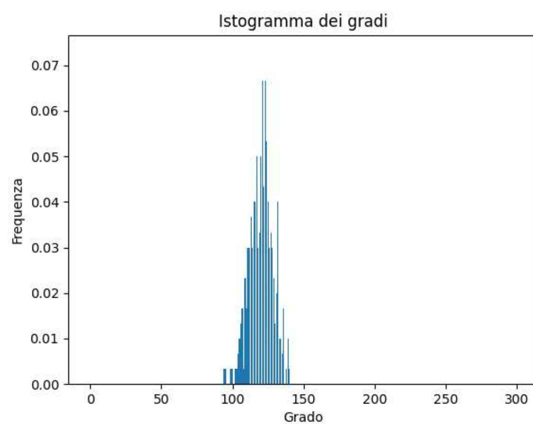

FIGURA 2.3: Istogramma dei gradi per un grafo aleatorio di parametri $n = 300$, $p = 0.4$.

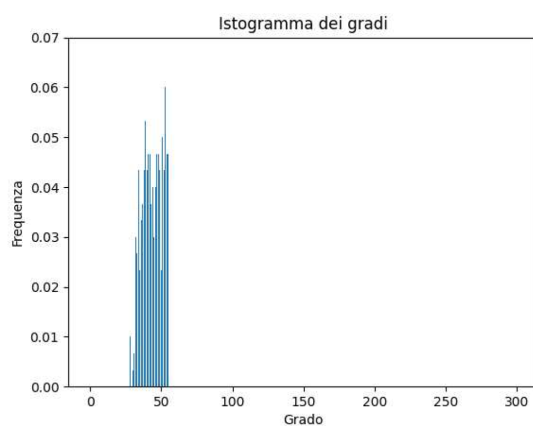

FIGURA 2.4: Istogramma dei gradi per un grafo aleatorio di parametri $n = 300$, $p = 0.4$ e grado massimo pari a 55.

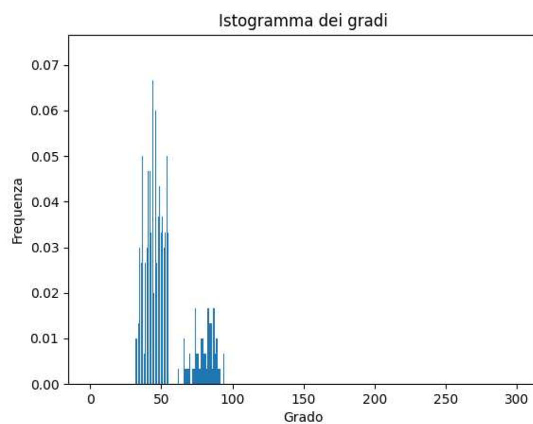

FIGURA 2.5: Istogramma dei gradi per un grafo aleatorio di parametri $n = 300$, $p = 0.4$, grado massimo pari a 55 e proporzione di nodi centrali pari a 0.2.

2.3.1 Funzione `compare_degree_distributions()`

Questa funzione è pensata per confrontare visivamente la distribuzione dei gradi dei nodi di alcuni grafi aleatori al variare della probabilità di connessione p, mantenendo costanti il parametro n e gli altri parametri opzionali. Sull'asse delle ascisse sono presenti i gradi dei nodi d_i mentre sull'asse delle ordinate i valori da 0 a 1 per la proporzione di nodi di grado d_i osservati nel singolo grafo. Questo grafico aiuta a capire come la probabilità p influenza la struttura del grafo rispetto alla distribuzione dei gradi. Nella funzione sono richiesti in input i parametri `n, max_degree, ratio_hub_nodes` ed un vettore `p_values` di valori p.

La Figura 2.6 mostra un esempio della distribuzione dei gradi di un grafo con 500 nodi e tre diversi valori di p, mentre la Figura 2.7 mostra un esempio della distribuzione dei gradi di un grafo con 500 nodi e tre diversi valori di p ed una limitazione dei gradi dei nodi.

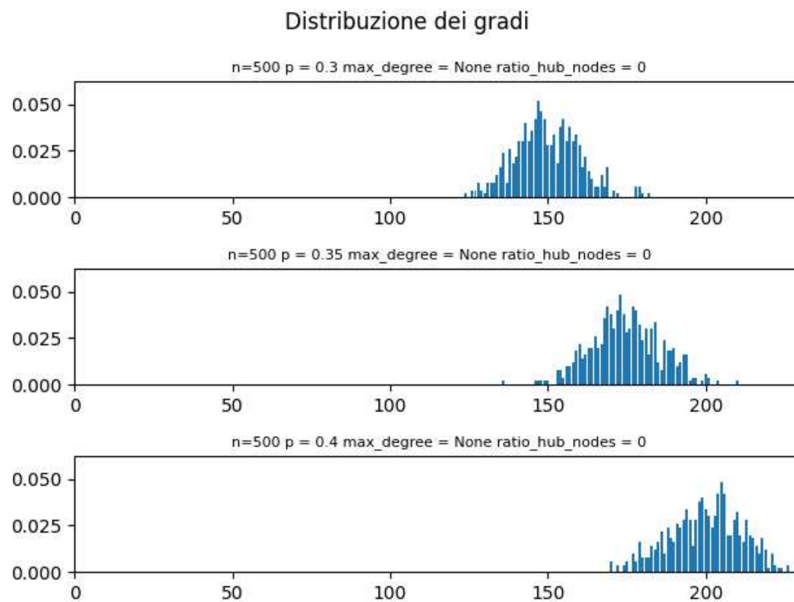

FIGURA 2.6: Istogramma dei gradi per un grafo aleatorio indiretto di parametri $n = 500$ e p rispettivamente pari a 0.30, 0.35, 0.40.

2.3.2 Funzione `plot_3d_property_frequency()`

Tale funzione ha lo scopo di creare grafici tridimensionali per esplorare la frequenza con cui si verificano determinate proprietà nei grafi aleatori presi in considerazione. Questi grafici possono essere utili per analizzare come la probabilità di osservare una determinata proprietà in un grafo varia in funzione di diversi valori di n e p, mantenendo costanti

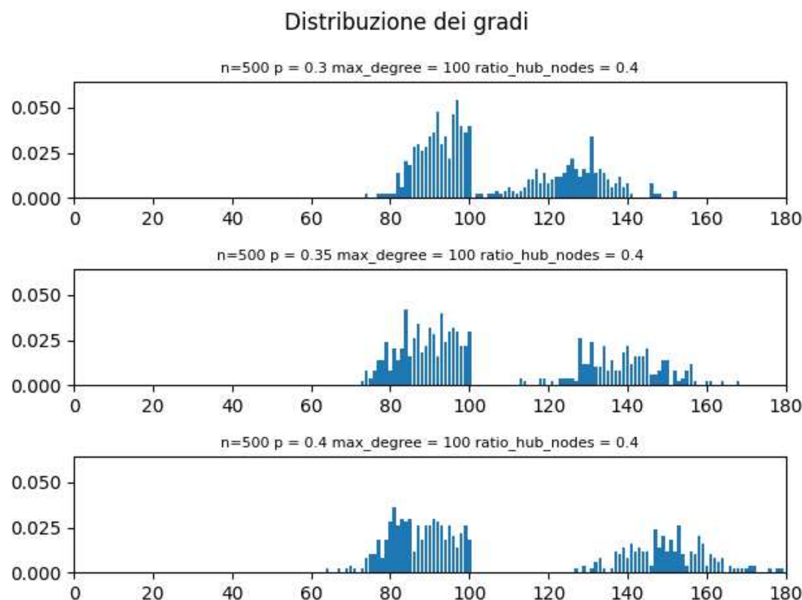

FIGURA 2.7: Istogramma dei gradi per un grafo aleatorio indiretto di parametri $n = 500$, $d_{max} = 100$ e $\pi = 0.4$ e p rispettivamente pari a 0.30, 0.35, 0.40.

i parametri di grado massimo e la proporzione di nodi centrali. La frequenza rappresentata è ottenuta dall'osservazione o meno della proprietà in campioni di numerosità `mc` per ogni combinazione di valori presenti nei vettori `n_list` e `p_list` dati in input. Oltre a questi parametri è richiesta in input la proprietà specifica da studiare scelta tra connessione, aciclicità e bipartizione, e gli usuali parametri opzionali `max_degree`, `ratio_hub_nodes`. Nel prosieguo del lavoro verranno forniti diversi esempi riguardo il funzionamento di questa funzione.

2.4 Funzioni di salvataggio e acquisizione da file

Le ultime due funzioni proposte sono utili nel caso in cui si voglia generare, per scopi di analisi o sperimentazione, un numero elevato di grafi aleatori con determinati parametri in input. Le due funzioni hanno lo scopo di salvare e riacquisire da file i campioni di grafi per poter utilizzare gli stessi dati più volte, evitando il costo computazionale di una generazione ripetuta di campioni (soprattutto se le strutture prese in considerazione hanno una numerosità di nodi sull'ordine delle centinaia, oppure sono necessarie importanti operazioni di abbassamento dei gradi) e rendendo possibile la replicazione dei risultati. Il codice citato sarà consultabile in Appendice: Listato 4.

2.4.1 Funzione `write_graphs_on_file()`

Questa prima funzione genera campioni di `mc` elementi per ogni combinazione di valori n e p presente nei vettori in input `p_list` e `n_list`, mantenendo fissi i parametri opzionali `n, max_degree, ratio_hub_nodes`. Se non viene specificato alcun nome di file `file_name` in cui salvare i dati, ciò avverrà automaticamente in `grafi.txt`.
Il formato con cui vengono salvate le informazioni è un file testuale di valori separati da spazio. Ciascun campione presenta una riga di intestazione con i parametri utilizzati nella generazione dei dati per ogni combinazione di valori in `p_list` e `n_list`, una riga di intestazione per ogni singolo grafo aleatorio etichettata da un intero compreso tra 0 e $mc - 1$, e la lista completa degli archi.

2.4.2 Funzione `read_graphs_from_file()`

Quest'ultima funzione apre il file di testo (se esistente) e ne salva i grafi in una struttura dati di tipo dizionario in cui ciascuna chiave è la stringa di parametri del campione di grafi considerato `[n, p, mg, rhn, mc]` (dove `md` sta per `max_degree` e `rhn` sta per `ration_hub_nodes`) e ciascun valore è la lista di grafi corrispondenti. Questa organizzazione facilita e velocizza l'accesso ai grafi contenuti nel file permettendo l'analisi dei dati in forma singola o aggregata.

2.5 Interfaccia grafica

Tutte le funzionalità elencate precedentemente sono messe a disposizione dell'utente attraverso un'interfaccia grafica intuitiva che permette un'interazione con il software agevolata rispetto che da riga di comando.

In questo progetto l'interfaccia è stata implementata attraverso la libreria `tkinter` di Python ed è organizzata in cinque sezioni visibili nelle Figure 2.8, 2.9, 2.10, 2.11, 2.11. La prima utile per la generazione di grafi singoli con parametri personalizzati, la possibilità di rappresentarli e la verifica delle proprietà di connessione, aciclicità e bipartizione. La seconda per la generazione ed il salvataggio di grafi su file nel caso di utilizzo futuro o condivisione, semplificando così la gestione dei dati e la condivisione delle informazioni. La terza si occupa della lettura degli elementi da file ed offre un'analisi statistica dei dati in cui gli utenti possono esplorare i singoli elementi, rappresentarli e verificarne le proprietà. La quarta analizza nello specifico le tre proprietà prese in considerazione, generando un grafico tridimensionale che rappresenta la probabilità di osservare la proprietà scelta in campioni di grafi con parametri personalizzati e variabili.

La quinta, ed ultima, permette agli utenti di confrontare la distribuzione dei gradi al variare delle probabilità di connessione dei nodi, enfatizzando le differenze e le somiglianze tra le frequenze dei gradi attraverso una loro comparazione istantanea.

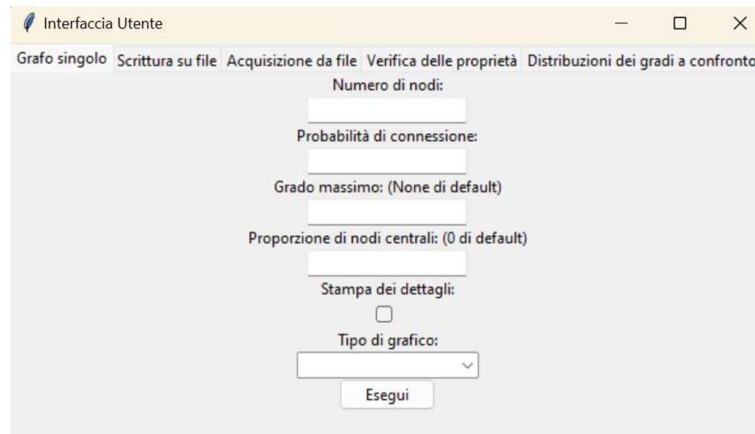

Figura 2.8: Sezione 1.

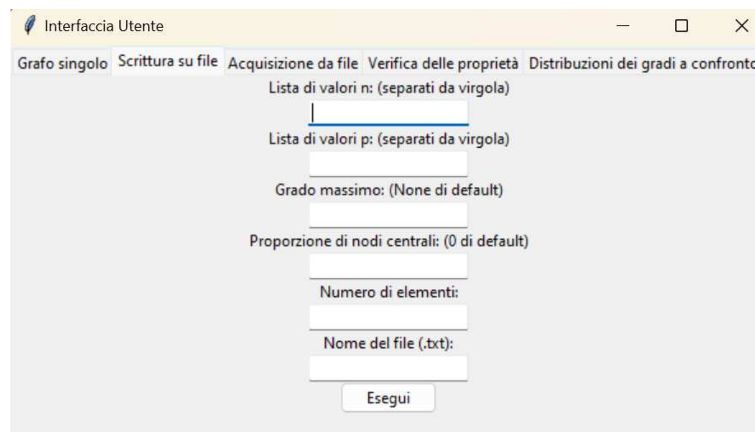

Figura 2.9: Sezione 2.

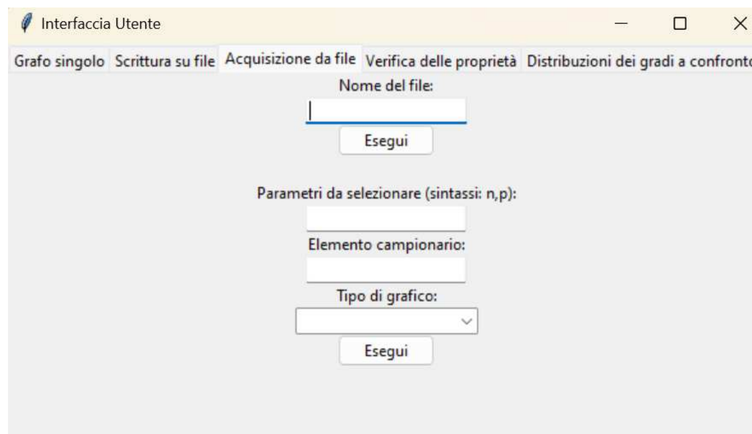

FIGURA 2.10: Sezione 3.

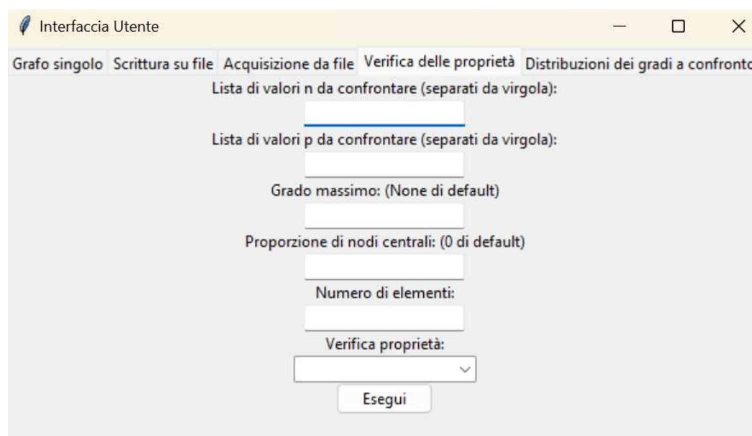

FIGURA 2.11: Sezione 4.

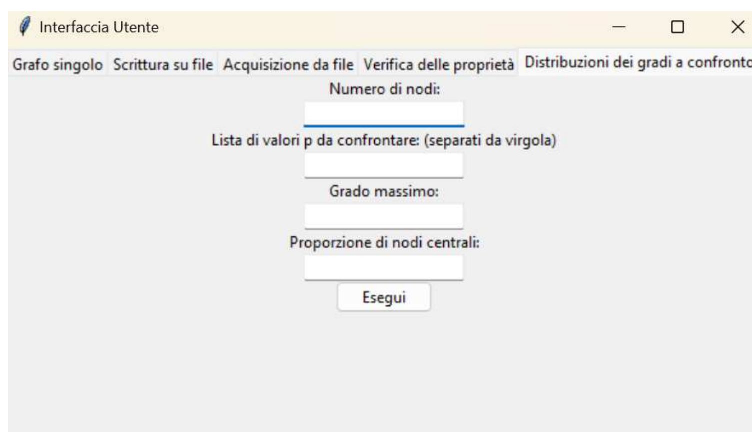

FIGURA 2.12: Sezione 5.

Capitolo 3

Conclusioni

In questo progetto, è stato sviluppato un programma per la generazione e l'esplorazione dei grafi aleatori con vertici di grado limitato, ottenendo un sistema che consente agli utenti di visualizzare diversi tipi di strutture dati, noti i parametri fondamentali, assumendo la completa randomicità delle conessioni. Il lavoro ha permesso, inoltre, di eseguire confronti di strutture dati di tipo diverso ed analisi esplorative sulle caratteristiche principali sulla distribuzione dei nodi o sull'osservazione delle proprietà prese in considerazione.

3.1 Risultati

Dopo l'esecuzione di alcune simulazioni, sono emersi dei risultati interessanti per quanto rigurda la distribuzione dei nodi. In particolare, attraverso gli istogrammi dei gradi, calcolati dapprima su singoli grafi e poi sul loro confronto, emerge che l'imposizione dei vincoli di grado genera distribuzioni che differiscono per forma, simmetria, curtosi e dispersione.

La distribuzione dei gradi $P(d)$ di partenza, per grafi aleatori senza imposizione di vincoli, è simmetrica e centrata attorno al valore di grado medio ottenuto tramite la formula

$$\overline{d} = d_{n-1}p \tag{3.1}$$

dove d_{n-1} è il grado massimo possibile per i nodi di un grafo di ordine n, e p la loro probabilità di connessione. Un esempio di questo è visibile in Figura 3.1 dove è chiaro osservare come la distribuzione dei gradi subisca una variazione di posizione nella direzione della variazione di p: se quest'ultima aumenta, aumentano anche la media e la mediana della distribuzione dei gradi $P(d)$.

La distribuzione $P(d)$ di grafi con vincolo di grado massimo, invece, risulta essere di forma variabile in funzione del grado massimo d_{max} impostato; se questo è pari ad un valore particolarmente elevato rispetto al grado medio $\overline{d}$, la curva subisce delle variazioni minime e la distribuzione rimane per lo più invariata. Se questo valore si avvicina al grado medio, o ne scende al di sotto, tutti i gradi superiori al massimo vengono ridotti, o al più azzerati, e la distribuzione assume una forma asimmetrica e troncata a destra in corrispondenza del grado massimo, con una densità completamente distribuita tra i valori 0 e d_{max}. Questo fatto è ben visibile nel grafico in Figura 3.2.

Se invece, oltre che all'imposizione del grado massimo, è presente anche un vincolo sulla proporzione di nodi centrali π, si osserva che viene a formarsi una distribuzione bimodale, risultato dalla mistura di due curve: una distribuzione asimmetrica troncata a destra in corrispondenza di d_{max} per i vertici con grado limitato, ed una distribuzione simmetrica per la frazione π di nodi centrali con grado arbitrario. Il divario tra le due è più o meno ampio in funzione della distanza che separa il grado medio $\overline{d}$ dell'ipotetico grafo aleatorio semplice corrispondente, cioè in assenza di vincoli, e del grado massimo impostato. La Figura 3.3 mostra un esempio di come, mantenendo invariato il parametro d_{max}, al crescere di p si accentui il divario tra la distribuzione troncata dei gradi dei nodi limitati e quella dei gradi dei nodi centrali, con una traslazione della seconda verso destra.

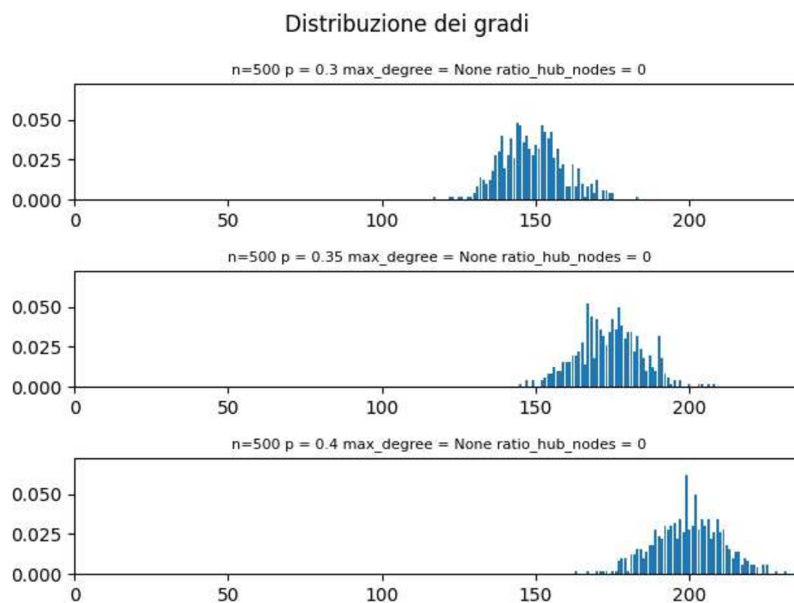

FIGURA 3.1: Istogramma dei gradi per un grafo aleatorio indiretto di parametri $n = 500$ e p rispettivamente pari a 0.30, 0.35, 0.40.

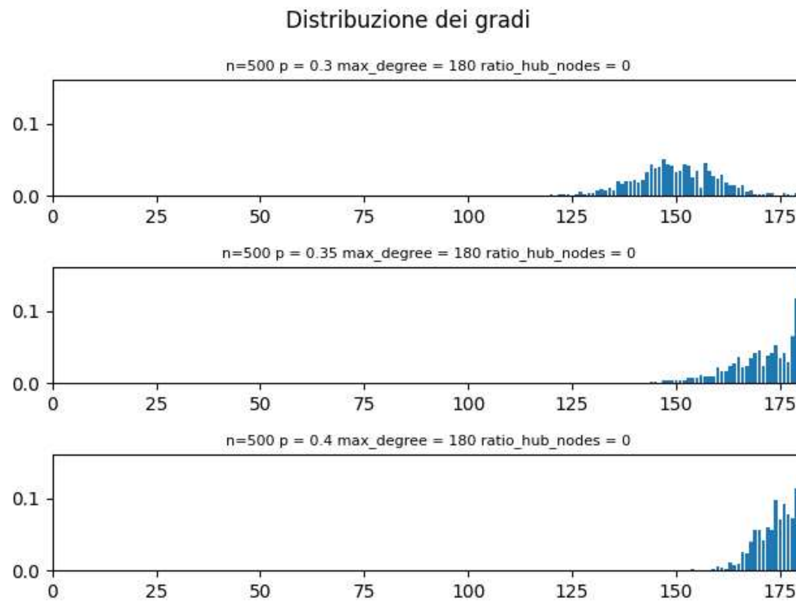

FIGURA 3.2: Istogramma dei gradi per un grafo aleatorio indiretto di parametri $n = 500$, $d_{max} = 180$ e p rispettivamente pari a 0.30, 0.35, 0.40.

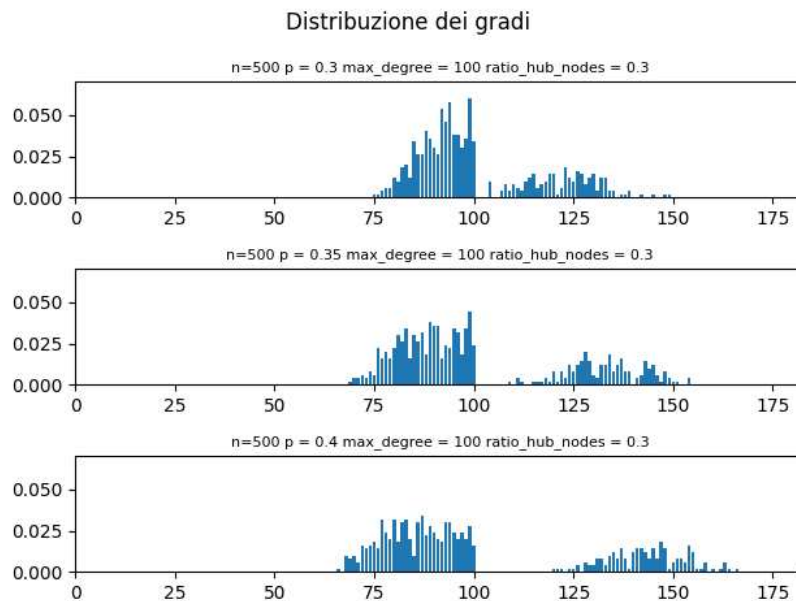

FIGURA 3.3: Istogramma dei gradi per un grafo aleatorio indiretto di parametri $n = 500$, $d_{max} = 100$, $\pi = 0.3$ e p rispettivamente pari a 0.30, 0.35, 0.40.

Ulteriori risultati interessanti sono stati osservati verificando le proprietà in campioni in campioni di grafi di numerosità 100 al variare dei parametri. Le considerazioni che seguono sono state ottenute da strutture dati con dei numeri di nodi dell'ordine del centinaio, compresi tra 100 e 135, e senza imposizione di vincoli sulla distribuzione dei gradi.

Volendo studiare la connettività (Figura 3.4) è emerso che si cominciano ad osservare

grafi connessi già a partire da una probabilità di esistenza degli archi $p = 0.03$, mentre per valori di $p > 0.10$ si può assumere una connessione quasi certa. Ciò è reso possibile dal fatto che gli archi sono uniformemente distribuiti all'interno della matrice di adiacenza e quindi, non essendo presenti processi di selezione che portano alla formazione di componenti centrali, raggiunta la probabilità $p = 0.10$ tutti i nodi potrebbero avere almeno un arco che li mette in comunicazione con gli altri, diventando così raggiungibili a partire da buona parte degli altri nodi del grafo. Si osservi che questa proprietà dipende per lo più da p e non, apparentemente, da n.

La situazione è molto diversa per quanto riguarda la verifica della mancanza di cicli all'interno dei grafi (Figura 3.5). Per osservare una variabilità dei risultati campionari, date le numerosità di nodi citate in precedenza, è necessario tenere una probabilità di connessione $p < 0.02$. Questo significa che la presenza di cicli si verifica più velocemente della connessione, dunque, non è necessario che un grafo sia connesso per poter osservare la presenza di cammini chiusi. Si nota, inoltre, che la numerosità campionaria non è più ininfluente come visto sopra, e la mancanza di cicli si presenta più difficilmente in grafi con ordine n elevato piuttosto che per grafi di ordine inferiore.

Un risultato molto simile al precedente si verifica anche nel caso della bipartizione. Anch'essa presenta dei risultati mutevoli per valori di $p < 0.02$ facendo intuire che le due proprietà (aciclicità e bipartizione) si possono osservare assieme: se un grafico presenta dei cicli tendenzialmente non è bipartito e, dunque, non è divisibile in due insiemi disgiunti tali che siano presenti archi solo tra nodi di insiemi differenti. Come sopra, la proprietà dipende non solo dalla probabilità di connessione p, ma anche dalla numerosità dei nodi n e si verifica maggiormente, a parità di p, in grafi di ordine inferiore.

3.2 Possibili sviluppi futuri

L'analisi esplorativa condotta fino adesso ha permesso di far emergere alcune osservazioni aprendo interessanti prospettive per eventuali sviluppi futuri. La prima tra queste rigurda le ipotesi di tipo distributivo sulla forma della densità $P(d)$: sarebbe interessante indagare con maggiore dettaglio le implicazioni dovute al variare dei parametri nella distribzuione dei nodi, cercando di individuare una distribuzione asintotica dei gradi dei nodi generata dal modello preso come riferimento, sia in caso di presenza che in caso si assenza dei vincoli. Alternativamente, potrebbe essere rilevante anche capire come la probabilità di connessione e l'imposizione dei vincoli influenzi la connettività, la formazione di cicli o la bipartizione, considerando un intervallo di probabilità p più ampio e studiando la convergenza per valori di n di ordine superiore.

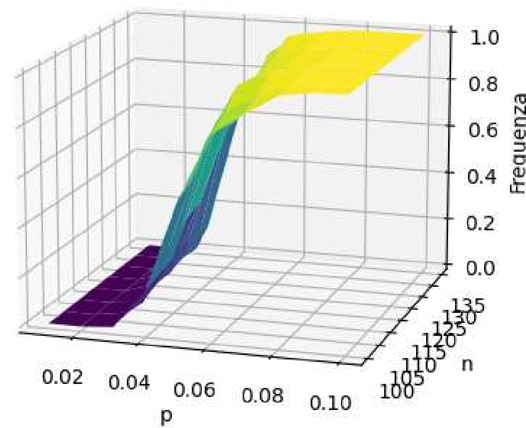

FIGURA 3.4: Proporzione di grafi aleatori connessi per campioni di numerosità pari a 100, con n variabile compreso tra 100 e 135 e p variabile compreso tra 0.01 e 0.1.

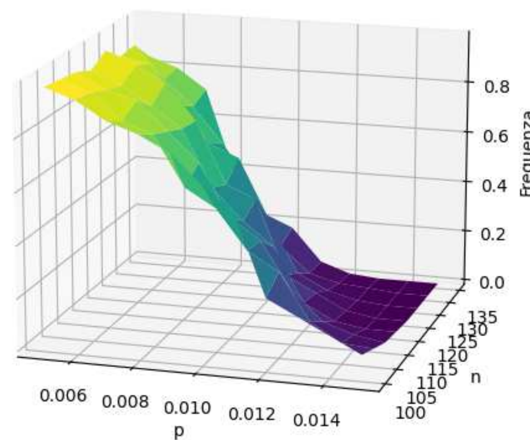

FIGURA 3.5: Proporzione di grafi aleatori aciclici per campioni di numerosità pari a 100, con n variabile compreso tra 100 e 135 e p variabile compreso tra 0.005 e 0.015.

Sviluppi più concreti e specifici si potrebbero avere sia confrontando queste simulazioni con reti complesse del mondo reale che sembrerebbero avere la stessa struttura, valutando quanto questi modelli di grafi aleatori con ordine di vertice limitato si avvicinino o si discostino dai dati reali, sia applicando il modello a contesti pratici o problemi specifici. Ad esempio, potrebbe esserci un'applicazione in domini come la modellazione di reti sociali, di reti di comunicazione o di reti biologiche. In caso affermativo, un passo successivo potrebbe essere quello di eseguire un'ottimizzazione degli algoritmi

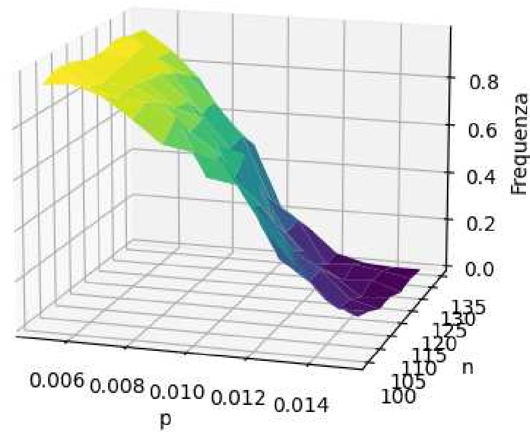

FIGURA 3.6: Proporzione di grafi aleatori bipartiti per campioni di numerosità pari a 100, con n variabile compreso tra 100 e 135 e p variabile compreso tra 0.005 e 0.015.

proposti, aumentandone l'efficienza computazionale e migliorando la gestione dei dati in memoria.

Appendice

Per l'esecuzione di questo programma sono state utilizzate diverse librerie di Python. Le tre principali che hanno contribuito alla realizzazione delle strutture dati implementate sono `NetworkX`, `NumPy` e `Matplotlib`. La prima, e più significativa, è un pacchetto Python per la creazione, la manipolazione e lo studio della struttura, della dinamica e delle funzioni di reti complesse fornendo, in questo contesto, le risorse necessarie per implementare i grafi in modo efficiente mediante le sue classi e funzionalità. Offre, inoltre, delle procedure che permettono di calcolare le statistiche descrittive e verificare le proprietà dei grafi. La seconda è una libreria scritta in C fondamentale per il calcolo scientifico. Essa mette a disposizione array multidimensionali e vari oggetti derivati, permettendo operazioni statistiche di base e la simulazione casuale. La terza è una libreria grafica cruciale in questo progetto perché, non solo ha reso possibile tutti i tipi di grafici presenti ma, attraverso i suoi strumenti di visualizzazione ha aiutato a comprendere e verificare i risultati prodotti. In aggiunta, sono state impiegati anche i pacchetti `time` per la misura dei tempi di esecuzione, `pprint` per la stampa formattata di oggetti come dizionari e matrici, e `tkinter` per la creazione dell'interfaccia standard mediatrice tra software e utente (3).

```python
import networkx as nx
import numpy as np
import time
import pprint
import matplotlib.pyplot as plt
from matplotlib.cm import Blues
import tkinter as tk
from tkinter import ttk
```

Listato 1: Librerie.

Segue il codice dei paragrafi 2.2, 2.3 e 2.4.

```
def draw_circular_graph(Graph):
    pos = nx.circular_layout(Graph)  #posizionamento dei nodi

    #aggiiunta nodi, etichette e archi
    nx.draw_networkx_nodes(Graph, pos, node_size=500, node_color='white', edgecolors='black')
    nx.draw_networkx_labels(Graph, pos, font_size=10, font_color='black')
    nx.draw_networkx_edges(Graph, pos, width=1.2, edge_color='black')

    plt.title("Grafo aleatorio indiretto") #titolo
    plt.axis('off') #rimozione degli assi cartesiani

    return plt.show()

def draw_subgraphs(Graph):
    forest = list(nx.connected_components(Graph)) #generazione di un insieme di nodi per ogni sottografo
    col_dict = {} #dizionario del colore dei nodi
    colors = [] #colori totali
    node_colors = [] #colore di ciascun nodo in relazione al sottografo

    #assegnamento di un colore dalla palette Blues per ogni sottografo
    for i in range(len(forest)):
        colors.append(Blues(0.25 + (i / len(forest))*0.75)) #intervallo dei colori ristretto per evitare tonalit  tendenti al bianco

    for i, subgraph in enumerate(forest): #indici di scorrimento dei grafi
        for node in subgraph:
            col_dict[node] = colors[i] #dizionario (nodo:colore)

    pos = nx.spring_layout(Graph)  #posizionamento dei nodi con enfatizzazione dei cluster

    for node in Graph.nodes():
        node_colors.append(col_dict[node]) #colorazione dei nodi

    nx.draw(Graph, pos, node_color=node_colors, with_labels=True)

    return plt.show()
```

```
36 def histogram_of_graph_degrees(Graph):
37
38     n_nodes = nx.number_of_nodes(Graph)
39     hist = nx.degree_histogram(Graph) #frequenze dei gradi assolute
40     hist = [x/n_nodes for x in hist] #frequenze relative
41
42     while len(hist) < (n_nodes):
43         hist.append(0) #completamento dell'istogramma con i gradi
   mancanti
44
45     plt.bar(range(n_nodes), hist, align='center') #grafico a barre
46     plt.ylim(0, max(hist)+0.01) #limitazione dell'asse y
47
48     #etichette e titolo:
49     plt.xlabel('Grado')
50     plt.ylabel('Frequenza')
51     plt.title('Istogramma dei gradi')
52
53     return plt.show()
```

Listato 2: Funzioni del paragrafo 2.2: Analisi grafica.

```
def compare_degree_distributions(n, p_values, max_degree = None,
    ratio_hub_nodes = 0):
    _, axs = plt.subplots(len(p_values), 1)  #impostazione della
    dimensione della figura

    plt.suptitle('Distribuzione dei gradi') #titolo
    max_y = 0 #valore massimo dell'asse y
    max_x = 0

    #generazione dei sottografici
    for i in range(len(p_values)):
        graph = random_graph_with_hub_nodes(n, p_values[i], max_degree
    , ratio_hub_nodes) #grafo aleatorio

        n_nodes = nx.number_of_nodes(graph)
        hist = nx.degree_histogram(graph)   #frequenze assolute dei
    gradi
        hist = [x / n_nodes for x in hist]  #frequenze relative

        #ricerca del limite superiore degli assi per avere finestre
    delle stesse dimensioni e rendere confrontabili le distribuzioni
        max_x = max(max_x, len(hist))
        max_y = max(max_y, max(hist))

        while len(hist) < (n_nodes):
            hist.append(0) #completamento dell'asse x con i gradi
    mancanti

        axs[i].bar(range(n_nodes), hist, align='center') #grafico a
    barre
        axs[i].set_title('n={} p = {} max_degree = {} ratio_hub_nodes
    = {}'.format(n, p_values[i], max_degree, ratio_hub_nodes), fontsize
     = 8) #inserimento dei parametri nel titolo

    for ax in axs: #limitazione degli assi
        ax.set_xlim(0, max_x)
        ax.set_ylim(0, max_y + 0.01)

    plt.tight_layout() #posizionamento dei sottografi

    return plt.show()


```

```
35 def connectivity(Graph): #funzione di verifica di della connessione
36     return nx.is_connected(Graph)
37
38 def aciclicity(Graph): #funzione di verifica di dell'aciclicita'
39     return nx.is_forest(Graph)
40
41 def bipartition(Graph): #funzione di verifica di della bipartizione
42     return nx.is_bipartite(Graph)
43
44 def plot_3d_property_frequency(n_values, p_values, mc = 40, max_degree
     = None, ratio_hub_nodes = 0, property = '', title = 'Frequenza'):
45
46     if property not in [connectivity, bipartition, aciclicity]: #
    diagnostica
47         print('Error: property string is not in\n[connectivity,
    bipartion, aciclicity].')
48         return None
49
50     n_values = sorted(n_values)
51     p_values = sorted(p_values)
52
53     freq = np.zeros((len(n_values), len(p_values))) #matrice di
    frequenze
54     one_sample = [0]*mc #valori osservati della cratteristica d'
    interesse per un campione di grafi con parametri n[i] e p[j]
55
56     for i in range(len(n_values)):
57         print(n_values[i]) #stato di avanzamento
58         for j in range(len(p_values)):
59             for k in range(mc):
60                 g = random_graph_with_hub_nodes(n = n_values[i], p =
    p_values[j], max_degree = max_degree, ratio_hub_nodes =
    ratio_hub_nodes)
61                 one_sample[k] = property(g) # 1 se il grafo k con
    parametri n[i], p[j] verifica la proprieta'
62
63             freq[i][j] = np.mean(one_sample) #numero di grafi che
    verificano la proprieta' all'interno del campione
64
65     #grafico: assi
66     x, y = np.meshgrid(p_values, n_values)
67     z = freq
68
69     fig = plt.figure()    #finestra di disegno
```

```
70      gr = fig.add_subplot(111, projection='3d') #sottografico 3-dim con
        griglia 1x1
71      gr.plot_surface(x, y, z, cmap='viridis')
72
73      #etichette degli assi e titolo
74      gr.set_xlabel('p')
75      gr.set_ylabel('n')
76      gr.set_zlabel('Frequenza')
77      gr.set_title(title)
78
79      return plt.show()
```

Listato 3: Funzioni del paragrafo 2.3: Confronto.

```python
def write_graphs_on_file(n_list, p_list, max_degree = None, ratio_hub_nodes = 0, mc = 40, file_name='grafi.txt'):
    start = time.time()

    n_list = sorted(n_list)
    p_list = sorted(p_list)

    with open(file_name, 'wb') as file: #apertura del file in scrittura
        for n in n_list:
            print('------------->', n) #stato di avanzamento

            if max_degree == None:
                md = n-1 #grado massimo senza vincoli
            else:
                md = max_degree

            for p in p_list:
                n = int(n)
                p = round(p, 2)
                file.write(f'{n} {p} {md} {ratio_hub_nodes} {mc}\n'.encode()) #scrittura dei parametri
                for k in range(mc):
                    file.write(f'{k}\n'.encode()) #scrittura dell'indice del grafo
                    g = random_graph_with_hub_nodes(n, p, max_degree, ratio_hub_nodes) #grafo
                    nx.write_edgelist(g, file, data=False) #scrittura dei nodi

    stop = time.time()
    print('Run time: ', (stop-start), 'seconds') #tempo di esecuzione
    return file_name

def read_graphs_from_file(file_name):
    try: #verifica di esistenza del file
        with open(file_name, 'rb') as file:
            g_dict = {}#dizionario di grafi (parametri:lista di grafi)
            g_list = [] #lista contenente gli mc grafi, una per ogni coombinazione di n, p
            g = None #singolo grafo
            n = None #valore corrente di n
            p = None #valore corrente di p
```

```
37
38             for l in file:
39                 l = l.decode().strip() #l = linea di file
40                 d = l.split() #d = digits, stringhe contenute su
    ciascuna riga
41
42                 if len(d) == 5: #intestazione di una lista di grafi
43
44                     if g is not None: #se esiste gia' una lista di
    grafi
45                         g_list.append(g) #salvataggio dell'ultimo
    grafo
46                         g_dict[(n, p, md, rhn, mc)] = g_list #aggiunta
     della lista al dizionario
47
48                     n, p, md, rhn, mc = map(float, d) #acquisizione
    parametri del nuovo campione
49                     g_list = [] #iniziaizzazione della lista di grafi
50                     g = None
51
52                 else:
53                     if len(d) == 1: #nuovo grafo singolo
54                         g_list.append(g)
55                         g = nx.Graph()
56
57                         for i in range(int(n)):
58                             g.add_node(i) #aggiunta n nodi
59
60                     if len(d) == 2:
61                         u, v = map(int, d) #aggiunta archi
62                         g.add_edge(u, v)
63
64             if n is not None and p is not None:
65                 g_dict[(n, p, md, rhn, mc)] = g_list #aggiunta della
    lista di grafi al dizionario
66
67         return g_dict
68
69     except FileNotFoundError: #se il file non esiste
70         return None
```

Listato 4: Funzioni del paragrafo 2.4: Salvataggio e acquisizione da file.

Bibliografia

[1] A. Vespignani A. Barrat, M. Barthélemy. *Dynamical Processes on Complex Networks*. Cambridge University Press, Cambridge, 2012.

[2] B. Bollobás. *Random Graphs*. Cambridge University Press, Cambridge, 2001.

[3] Python Software Foundation. *Python 3.10 documentation*, 2001-2023. Python version 3.10.4.

[4] Z.Petrášek M. Kang. Random graphs: Theory and applications from nature to society to the brain. *Internat. Math. Nachrichten*, 227:1–24, 2014.

[5] A. Rényi P. Erdos. On thr evolution of random graphs. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, 5:17–61, 1960.