



Esame IA: Intelligent Models

Relazione Progetto:

Epidemic Super Spreader in Networks

Studente: Ludovico Guercio

Matricola: 340036

DIPARTIMENTO DI MATEMATICA E INFORMATICA,
UNIVERSITÀ DEGLI STUDI DI PERUGIA

Indice

1	Obbiettivo	1
2	SIR Model	1
2.1	Implementazione Sistema	1
2.2	Nodi Super-Spreader	2
3	Codice del programma	3
3.1	Modulo Sir.py	3
3.1.1	Metodo costruttore	3
3.1.2	Getters	5
3.1.3	Metodo draw_network	6
3.1.4	Metodo della simulazione	7
3.2	Individuazione Nodi Super-Spreader	9
3.2.1	Metodo spreaders_count	10
3.2.2	centralitynodes.py	10
3.3	Notebook SimulazioneSIR.ipynb	13
4	Simulazione epidemica	15
4.1	Simulazioni singole	16
4.2	Random Simulation	19

1 Obbiettivo

L'obbiettivo della traccia del progetto è quello di implementare un sistema di "epidemic diffusion" simulandolo in una rete complessa per poi determinare i nodi "**super spreader**".

2 SIR Model

Il sistema consiste in una variazione basata sul modello SIR (Susceptible - Infected - Recovered). In questi modelli infatti, data una popolazione, si va a simulare la diffusione di una epidemia (ad esempio di una malattia) durante un periodo di tempo. Ad ogni periodo di tempo, la diffusione è calcolata in base ai dei parametri.

In una rete inizialmente tutti nodi sono nello stato di "Susceptible" e possono successivamente diventare "Infected", e poi "Recovered"; il cambiamento di stato è condizionato dai parametri di definizione e condizione dell'epidemia.

2.1 Implementazione Sistema

Il sistema implementato lavora nel seguente modo: dato un file csv, che descrive i nodi e gli archi della rete, si va a generare una network iniziale; questa, poi, viene elaborata per eseguire successivamente la simulazione epidemico.

L'elaborazione iniziale permette di impostare lo stato di partenza della rete per la simulazione.

Inizialmente tutti i nodi vengono segnati nello stato di "Susceptibile" e di "Infected"; naturalmente, una parte dei nodi della rete vengono settati infetti permettendo di eseguire la simulazione. I nodi infetti iniziali sono definiti tramite un numero totale di tale nodi e assegnati casualmente.

Il passaggio di stato di un nodo è definito come segue:

Modello SIRS

- **Susceptible – > Infected**
- **Infected – > Recovered**
- **Recovered – > Susceptible**

Come anticipato, i passaggi di stato dei nodi sono dipendenti da diversi parametri:

- **p_trans**: indica la probabilità di transizione della malattia da un nodo **Infected** ad un altro **Susceptible**. Più la probabilità è alta, più un nodo che è connesso ha un altro infetto, ha più probabilità di contrarre la malattia
- **t_rec**: indica il tempo di recupero dalla malattia; una volta passato questo tempo, il nodo torna nello stato di **Recovered**.
- **t_sus**: indica il tempo per cui un nodo **Recovered** torni allo stato di **Susceptible**; una volta tornato in questo stato, il nodo non è più immune alla malattia
- **t_sim**: indicata il tempo totale della simulazione del sistema. Si utilizzano valori **integer** simulando che essi rappresentano i giorni

Modello SIR

- **Susceptible – > Infected**
- **Infected – > Recovered**

Il sistema permette di analizzare anche il caso speciale per cui un nodo non torni allo stato di "Susceptible", quindi rimanendo immune per sempre alla malattia. In tal caso si lavorerà con il parametro **t_sus** tendente all'infinito:

2.2 Nodi Super-Spreader

Il programma permette di poter individuare i nodi "Super Spreader" di una determinata rete attraverso una serie di simulazioni randomiche.

I nodi **super spreader** sono quei nodi che hanno la maggiore capacità e tendenza a comunicare con più nodi. Nel caso di tale sistema, si intende la facilità per cui un nodo riesce facilmente a trasmettere la malattia ad altri nodi; solitamente questi nodi sono anche considerati come nodi "hub", mentre il resto di nodi hanno capacità drasticamente minori di trasmissione.

L'individuazione di questi nodi super spreader, posto come obiettivo finale, viene fatto tramite un meccanismo preciso implementato nella classe **SIR** del modulo python **Sir.py**. Questo metodo permette di tenere traccia dei nodi che trasmettono la malattia durante le simulazioni; i nodi che infettano di più e con maggiore frequenza saranno i nodi super spreaders. Per una maggiore accuratezza, i risultati possono essere confrontati con le misure di centralità dei nodi della rete per controllare se i nodi super-spreaders corrispondono anche ai nodi "hub".

3 Codice del programma

Il programma è struttura principalmente su 3 file python:

- **Sir.py**
- **centralitynodes.py**
- **simulationSIR.pynb**

Nel modulo **Sir.py** viene implementata una classe denominata **Sir**, la quale descrive le caratteristiche e funzionalità per inizializzare e simulare una rete.

Con il modulo **centralitynodes.py** possiamo utilizzare i metodi che calcolano le misure di centralità dei nodi; utili se vogliamo avere un ulteriore confronto sull'analisi dell'importanza dei nodi e dei nodi influenti.

Il **simulationSIR** è un notebook python utilizzato per le simulazioni, ad esempio, in Google Colab per un migliore uso di librerie grafiche come *NetworkX* (per i grafi e reti), *matplotlib* (per grafici, plotting ecc.)

3.1 Modulo Sir.py

Rappresenta il modulo python in cui viene definito il funzionamento vero e proprio del sistema SIR. Al suo interno è definita una classe **Sir**.

La classe **Sir** descrive quell'oggetto che permette di inizializzare una rete per il modello simulativo, visualizzare le caratteristiche e metriche, effettuare simulazioni.

3.1.1 Metodo costruttore

```
class Sir:  
    def __init__(self, init_infect, filename, delimiter): #init infect = numero int di nodi infetti  
        df = pd.read_csv('/content/' + filename, delimiter) #delimiter di default = ','  
        self.graph = nx.from_pandas_edgelist(df, source='from', target='to')  
        self.init_infect = init_infect  
        self.init_state = self.set_init_infect()
```

Il metodo `__init__` permette di inizializzare il grafo della rete per il modello simulativo. Vengono passati come parametri:

- `init_infect`: numero di nodi iniziali infetti
- `file`: il nome del file csv della rete
- `delimiter`: separatore dei valori nel file csv; di default è settato come uno spazio

Dati i parametri, si apre il file csv e si interpreta i dati al suo interno che descrivono gli archi dei nodi; quindi viene poi trasformato il csv in un grafo attraverso `NetworkX`. Infine si applica il metodo `set_init_infect()` per settare i nodi inizialmente infetti.

```
def set_init_infect(self):
    #get graph nodes
    nodes = self.graph.nodes
    sample = rd.sample(nodes, self.init_infect) #assegno tot nodi casuali come infetti

    #set parametri iniziali: infetti iniziali
    for node in nodes:
        #set initial infected
        if node in sample:
            nodes[node]['State'] = 'I'
            nodes[node]['T_rec'] = 0
            nodes[node]['T_sus'] = 0
            nodes[node]['DNA'] = [node]
            nodes[node]['INIT'] = True
        else:
            nodes[node]['State'] = 'S'
            nodes[node]['T_rec'] = 0
            nodes[node]['T_sus'] = 0
            nodes[node]['DNA'] = []
            nodes[node]['INIT'] = False

    #self.draw_network()
    infected = self.getInfected()
    print(f"Nodi Tot. Grafo:{len(nodes)}")
    print(f"Nodi Infetti Iniziali:{len(infected)}")
    print(f":{self.getInfected()}")


#self.draw_network()
infected = self.getInfected()
print(f"Nodi Tot. Grafo:{len(nodes)}")
print(f"Nodi Infetti Iniziali:{len(infected)}")
print(f":{self.getInfected()}")
```

`set_init_infect()` assegna randomicamente i nodi infetti iniziali sul totale posto come parametro iniziale. Per far ciò, per ogni nodo nell'array `sample` contenente i nodi da infettare, modifica gli attributi di tali nodi del grafo assegnandogli lo stato di **"I"** di infetto, **T_rec** e **T_sus** uguale a 0. Per il resto dei nodi del grafo, invece, gli viene attribuito lo stato di **"S"** susceptible e con tempo di recupero e di ritorno alla vulnerabilità sempre uguali a 0.

Inoltre per i nodi **"Infected"** viene assegnato il nodo iniziale infetto, così da etichettare il nodo **"DNA"** iniziale dell'epidemia; viene segnato anche l'attributo **"INIT"** a `True`, mentre per il resto dei nodi viene settato a `False`. L'attributo del nodo **DNA** permette di memorizzare tutti i nodi di cui un nodo ha ricevuto la malattia: in sostanza permette di avere un registro di etichette dei nodi infetti per tenere traccia dei dei nodi spreaders. Inoltre a livello grafico, i nodi **Infetti** sono colorati di **rosso**, i nodi **Suscettibili** di **celeste**, mentre i nodi **Guariti** di **verde**.

Ad esempio, già subito con questi due metodi, possiamo definire un oggetto SIR che rappresenta una nuova rete inizializzata con i parametri di nodi infetti iniziali:

```
network = Sir(40, 'network.csv', delimiter = ' ')
#oppure
network = Sir(40, 'network.csv', delimiter = ',', )
```

3.1.2 Getters

All'interno della classe ho definito dei metodi **getters**, utili per poter ritornare le informazioni e metriche relative alla rete e agli stati dei loro nodi:

- `getNeighbors`: dato un nodo, ritorna una lista di tutti i suoi nodi vicini
- `getNeighbors_Sus`: dato un nodo, ritorna una lista di tutti i suoi nodi vicini che hanno lo stato di "Suscettibile"
- `getSusceptible`: ritorna la lista di tutti i nodi nello stato di "Suscettibile"
- `getInfected`: ritorna la lista di tutti i nodi nello stato di "Infetto"
- `getRecovered`: ritorna la lista di tutti i nodi nello stato di "Guariti"
- `whoInfects`: ritorna la lista di tutti i nodi infettati da un certo nodo dato in input

```
def getNeighbors(self, node):
    return [n for n in self.graph.neighbors(node)]

def getNeighbors_Susceptibile(self, node):
    neigh_susceptible = []
    for n in self.graph.neighbors(node):
        if self.graph.nodes[n]['State'] == 'S':
            neigh_susceptible.append(n)

    return neigh_susceptible

def getSusceptible(self):
    susceptible = []
    for n in self.graph.nodes:
        if self.graph.nodes[n]['State'] == 'S':
            susceptible.append(n)

    return susceptible

def getInfected(self):
    infected = []
    for n in self.graph.nodes:
        if self.graph.nodes[n]['State'] == 'I':
            infected.append(n)

    return infected
```

```

def getRecovered(self):
    recovered = []
    for n in self.graph.nodes:
        if self.graph.nodes[n]['State'] == 'R':
            recovered.append(n)

    return recovered

def whoInfects(self, node):
    infects = []
    for n in self.graph.nodes:
        for nd in self.graph.nodes[n]['DNA']:
            if nd == node:
                infects.append(n)

    return infects

```

3.1.3 Metodo draw_network

Questa funzione d'utilità permette, dato un grafo di una rete, di riportare a video il grafo della rete con tutti i stati dei nodi. Richiamando i metodi `getters`, si recuperano la liste dei nodi suddivisi per i loro stati, che vengono poi colorati secondo i loro colori. Si utilizza per la stampa i metodi della libreria di `NetworkX`, `nx.spring_layout()`, `nx.draw_networkx_nodes()`, `nx.draw_networkx_edges()`

```

#draw the network with node states
def draw_network(self):
    color_map = []
    susceptible = self.getSusceptible()
    infected = self.getInfected()
    recovered = self.getRecovered()

    for s in susceptible:
        color_map.append('■#abcdef') # colore celeste

    for i in infected:
        color_map.append('■#ff5349') #colore rosso

    for r in recovered:
        color_map.append('■#98ff98') #colore verde

    plt.figure(figsize=(20,10))
    pos = nx.spring_layout(self.graph, iterations = 15, seed=1721)
    nx.draw_networkx_nodes(self.graph, pos, node_size=100, node_color=color_map)
    nx.draw_networkx_edges(self.graph, pos, alpha=0.2)
    plt.axis('off')
    plt.show()

```

3.1.4 Metodo della simulazione

Il metodo **simulation** è quello che permette di eseguire la simulazione nella nostra rete e andare a determinare i nodi super spreaders. **Prende come input il grafo della rete e i parametri personalizzabili di simulazione.** Come descritto in precedenza nelle generalità del progetto, questi parametri sono:

- **p_trans**: indica la probabilità di transizione della malattia da un nodo **Infected** ad uno **Susceptibile**
- **t_rec**: indica il tempo di recupero dalla malattia da parte di un nodo **Recovered**
- **t_sus**: indica il tempo per cui un nodo **Recovered** torni allo stato di **Susceptible**
- **t_sim**: indicata il tempo totale della simulazione del sistema.

Esempio utilizzo del metodo di simulazione:

```
simulazione = network.simulation(0.05, 7, 20, 60)

# dove p_trans = 0.05 , t_rec = 7, t_sus = 20, t_sim = 60
```

La simulazione all'interno della rete avviene come segue:

conoscendo il tempo totale di simulazione, per ogni intervallo di questo, quindi i giorni, si verifica il cambiamento di stato da parte dei nodi: ad ogni istante di tempo t , si recuperano tutti i nodi con i loro attributi di stato e parametri di tempo:

```
#method to start a single simulation
def simulation(self, p_trans, t_rec, t_sus, t_sim):

    G = self.graph
    #Initial/live/final data list per grafici e statistiche
    tot_S = []
    tot_I = []
    tot_R = []
    Timeline = [] #array temporale per grafico e statistiche

    for t in range(1,t_sim+1):
        Timeline.append(t)
        #get nodes
        susceptible = self.getSusceptible()
        infected = self.getInfected()
        recovered = self.getRecovered()
```

La simulazione della diffusione della malattia viene effettuata tramite operazioni e controlli per ogni tipo di nodo:

- **Infezione:** per ogni nodo nello stato di **Infected** si recuperato tutti i suoi nodi vicini suscettibili grazie al metodo `getNeighbors_Sus`; questi nodi essendo vicini degli infetti, possono contrarre la malattia. Se `p_trans` è maggiore di una possibile probabilità di contatto tra i due nodi, definita randomicamente, **il nodo suscettibile vicino passa allo stato di infected**. Inoltre il nodo "Infected" che trasmette la malattia viene registrato nell'attributo `DNA` del nodo così da ottenere delle etichette dei nodi che infettano
- **Guarigione:** Per ogni nodo nello stato di **Infected**, se il suo `t_rec` è maggiore o uguale al tempo `t_rec` impostato per la simulazione, allora il nodo è guarito, quindi **passa allo stato di recovered**; altrimenti il suo `t_rec` aumenta di 1
- **Vulnerabilità:** per ogni nodo nello stato di **Recovered**, passato un tempo `t_sus` maggiore o uguale di `t_sus` impostato per la simulazione, il nodo torna ad essere vulnerabile alla malattia, quindi **ritorna allo stato di susceptible**; altrimenti il suo `t_sus` aumenta di 1

```
for t in range(1,t_sim+1):
    Timeline.append(t)
    #get nodes
    susceptible = self.getSusceptible()
    infected = self.getInfected()
    recovered = self.getRecovered()

    for i in infected:
        neighbors = self.getNeighbors_Susceptible(i)
        for n in neighbors:
            #set random probability for that node
            p = np.random.rand()
            if p <= p_trans:
                G.nodes[n]['State'] = 'I'
                G.nodes[n]['T_rec'] = 0
                G.nodes[n]['T_sus'] = 0
                G.nodes[n]['DNA'].append(i)
```

```

#guarigione
for j in infected:
    if G.nodes[j]['T_rec'] >= t_rec:
        G.nodes[j]['State'] = 'R'
        G.nodes[j]['T_sus'] = 0
        G.nodes[j]['T_rec'] = 0
    else:
        G.nodes[j]['T_rec'] += 1

#ritornare ad essere suscettibili
for r in recovered:
    if G.nodes[r]['T_sus'] >= t_sus:
        G.nodes[r]['State'] = 'S'
        G.nodes[r]['T_rec'] = 0
        G.nodes[r]['T_sus'] = 0
    else:
        G.nodes[r]['T_sus'] += 1

```

La funzione termina con la definizione di due output: il primo output è un grafico creato attraverso `matplotlib` che riporta le metriche dell'andamento e variazione degli stati dei nodi durante il tempo simulativo.

L'altro output corrisponde al disegno grafico delle rete sociale una volta terminata la simulazione, questo grazie alla funzione `draw_network()`.

3.2 Individuazione Nodi Super-Spreader

I nodi **super-spreader** hanno una cruciale importanza all'interno di reti complesse, soprattutto quando si analizzano trasmissione di informazioni all'interno della rete, come nel contesto del progetto di diffusione epidemica.

L'obiettivo è quello di implementare una tecnica per individuare questi nodi nella rete.

La soluzione implementata consiste nell'etichettare i nodi che trasmettono la malattia nei nodi a loro volta infettati; in questo modo si avrà un registro completo dove poter andare a conteggiare quante volte questi nodi infettano i nodi della rete.

3.2.1 Metodo spreaders_count

Il meccanismo di individuazione dei nodi super spreaders è implementato nel metodo **spreaders_count**:

```
def spread_count(self): #to start after a simulation

    nodes = self.graph.nodes
    nodes_count = {}

    for n in nodes:
        for node in nodes[n]['DNA']:
            if node in nodes_count:
                nodes_count[node] += 1
            else:
                nodes_count[node] = 1

    df = pd.DataFrame.from_dict({
        'NODE': list(nodes_count.keys()),
        'SPREAD_COUNTS': list(nodes_count.values())
    })
    sort_counts = df.sort_values('SPREAD_COUNTS', ascending=False)
    sort_counts.index = np.arange(1, len(sort_counts) + 1)
    sort_counts.index.name = 'RANK'
    top_spreader = sort_counts[:15]
    print(tabulate(top_spreader, headers='keys', tablefmt='psql'))

    return top_spreader
```

Per ogni nodo della rete, si vanno a conteggiare i nodi presenti nell'attributo **DNA** e i risultati vengono tutti uniti in un nuovo dataframe pandas: qui avviene il conteggio dei migliori nodi che vengono classificati in base alla loro occorrenza; infine vengono portati in output tramite una tabella.

3.2.2 centralitynodes.py

In questo modulo python vengono utilizzati gli algoritmi di misura della centralità dei nodi, adatti per modelli simulativi epidemici.

Questi algoritmi possono essere utilizzati per il confronto con i risultati della simulazione.

- **Betweness Centrality:** misura l'importanza di un nodo all'interno di una shortest-path. Più questo nodo è presente in più shortest-path, maggiore è il suo valore di centralità.

Dato un nodo V, si calcola il numero **P** di shortest-path, tra i e j che lo attraversano; si calcola poi il totale **T** di shortest-path e poi si calcola il **rapporto tra P e T**; si esegue tutto per ogni coppia di nodi i e j.

- **Closeness Centrality**: misura le shortest per ogni coppia di nodi nella rete. Rappresenta una misura di velocità di trasmissione da un nodo all'altro; un alto valore, indica che l'informazione (o in questo caso la malattia) arriva più velocemente in tutta la rete. Si calcola con il reciproco della lunghezza media delle shortest-path da un nodo verso tutti gli altri.
- **Eigenvector Centrality**: misura la centralità tra un nodo e i suoi vicini; nodi influenti sono quelli vicini a nodi con valore di eigenvector elevato. Nodi acquisiscono importanza se sono vicini ad altri nodi importanti.
- **Degree Centrality**: misura la centralità di un nodo in base al numero dei suoi nodi vicini.
- **VoteRank Centrality**: simile a PageRank di Google: ogni nodo, per ogni turno, vota il suo nodo spreader influente, il nodo che riceve più voti per ogni turno diventa quindi un nodo-spreader; in base al numero di voti, si realizza il rank dei nodi votati.

```

def centrality_metrics(graph):
    dc = nx.degree_centrality(graph)
    cc = nx.closeness_centrality(graph)
    bc = nx.betweenness_centrality(graph)
    ec = nx.eigenvector_centrality(graph)
    vr = nx.voterank(graph, 15) #primi 15 più votati

    #from dict to pandas df
    dc_df = pd.DataFrame.from_dict({
        'node': list(dc.keys()),
        'degree centrality': list(dc.values())
    })

    cc_df = pd.DataFrame.from_dict({
        'node': list(cc.keys()),
        'closeness centrality': list(cc.values())
    })

    bc_df = pd.DataFrame.from_dict({
        'node': list(bc.keys()),
        'betweenness centrality': list(bc.values())
    })

    ec_df = pd.DataFrame.from_dict({
        'node': list(ec.keys()),
        'eigenvector centrality': list(ec.values())
    })

    vr_df = pd.DataFrame({'node':vr})

```

```

#from dict to pandas df
dc_df = pd.DataFrame.from_dict({
    'node': list(dc.keys()),
    'degree centrality': list(dc.values())
})

cc_df = pd.DataFrame.from_dict({
    'node': list(cc.keys()),
    'closeness centrality': list(cc.values())
})

bc_df = pd.DataFrame.from_dict({
    'node': list(bc.keys()),
    'betweenness centrality': list(bc.values())
})

ec_df = pd.DataFrame.from_dict({
    'node': list(ec.keys()),
    'eigenvector centrality': list(ec.values())
})

vr_df = pd.DataFrame({'node':vr})

def top_centrality_node(graph):
    #get df centrality metrics
    dc, cc, bc, ec, VR = centrality_metrics(graph)
    #sort top 10 and combine dataframe
    ec_sorted = ec.sort_values('eigenvector centrality', ascending=False)
    ec_sorted.index = np.arange(1, len(ec_sorted) + 1)

    cc_sorted = cc.sort_values('closeness centrality', ascending=False)
    cc_sorted.index = np.arange(1, len(cc_sorted) + 1)

    bc_sorted = bc.sort_values('betweenness centrality', ascending=False)
    bc_sorted.index = np.arange(1, len(bc_sorted) + 1)

    dc_sorted = dc.sort_values('degree centrality', ascending=False)
    dc_sorted.index = np.arange(1, len(dc_sorted) + 1)
    #voterank già ordina per importanza valore(rank), quindi setto solo gli index
    VR.index = np.arange(1, len(VR) + 1)

    BC = bc_sorted['node'][:15]
    CC = cc_sorted['node'][:15]
    EC = ec_sorted['node'][:15]
    DC = dc_sorted['node'][:15]

    ss = pd.merge(BC, CC, left_index=True, right_index=True)
    ss = pd.merge(ss, EC, left_index=True, right_index=True)
    ss = pd.merge(ss, DC, left_index=True, right_index=True)
    ss = pd.merge(ss, VR, left_index=True, right_index=True)
    ss.columns = ['BC','CC','EC','DC','VR']
    ss.index.name = 'RANK'
    print(tabulate(ss, headers='keys', tablefmt='psql'))

```

Le misure di centralità dei nodi possono essere anche visualizzate graficamente attraverso l'utilizzo di **heatmap** che evidenziano l'importanza di questi nodi direttamente gli "hotspot" nel grafo della rete.

```

def heatmap_centrality(G, metrics_name):
    cm = {}
    #get metrics
    if metrics_name == 'degree centrality':
        cm = nx.degree_centrality(G)
    if metrics_name == 'closeness centrality':
        cm = nx.closeness_centrality(G)
    if metrics_name == 'betweenness centrality':
        cm = nx.betweenness_centrality(G)
    if metrics_name == 'eigenvector centrality':
        cm = nx.eigenvector_centrality(G)

    df = pd.DataFrame.from_dict({
        'node': list(cm.keys()),
        metrics_name: list(cm.values())
    })
    sort_metrics = df.sort_values(metrics_name, ascending=False)
    top_metrics = sort_metrics[:15]

    plt.figure(figsize=(16,8))
    pos = nx.spring_layout(G, iterations= 15, seed=1721)
    nodes = nx.draw_networkx_nodes(G, pos, node_size=40, cmap=plt.cm.plasma,
                                   node_color=list(cm.values()),
                                   nodelist=cm.keys())
    nodes.set_norm(mcolors.SymLogNorm(linthresh=0.01, linscale=1, base=10))
    edges = nx.draw_networkx_edges(G, pos, alpha=0.2)

    plt.title(metrics_name)
    plt.colorbar(nodes)
    plt.axis('off')
    plt.show()
    print("Top 15:")
    print(top_metrics)

```

3.3 Notebook SimulazioneSIR.ipynb

Il file python notebook `simulazioneSIR.ipynb` viene implementato in Google Colab per poter simulare e visualizzare al meglio gli output grafici della rete e delle sue statistiche.

Nel notebook possiamo importare:

- il modulo networkSIR.py: descrive la classe **Sir** che permette di effettuare la simulazione
- il modulo centralitynodes.py
- i file csv: contengono i nodi e archi delle reti

SimulationSIR.ipynb

```

** IMPORT DEL MODULO PYTHON **

from google.colab import files
src = list(files.upload().values())[0]
open('NetworkSIR.py', 'wb').write(src)

```

```

** IMPORT FILE CSV **

uploaded = files.upload()

** IMPORT DELLA CLASSE SIR **

from NetworkSIR import SIR

```

Nel notebook viene implementato il metodo per applicare le simulazioni:

```

import pandas as pd
import numpy as np
from tabulate import tabulate

# Run random simulation and collect the node super spreaders for all
# simulation
def random_simulation(init_infect, filename, delimiter, p_trans, t_rec,
                      t_sus, t_sim, num_sim):

    networks = []
    spreaders_df = pd.DataFrame()

    if num_sim > 1:
        for i in range(num_sim):
            networks.append(Sir(init_infect, filename, delimiter))

    for network in networks:
        spread_count = network.simulation(p_trans, t_rec, t_sus, t_sim)
        spreaders_df = spreaders_df.append(spread_count)

    top_spreaders = spreaders_df.groupby(['NODE'])['SPREAD_COUNTS'].
    count().reset_index(
        name='FREQUENCY').sort_values(['FREQUENCY'], ascending=False)

    top_spreaders.index = np.arange(1, len(top_spreaders) + 1)
    top_spreaders.index.name = 'RANK'
    top_15 = top_spreaders[:15]
    print(tabulate(top_15, headers='keys', tablefmt='psql'))
else: #else if is only 1 sim, we get the draw graph output
    sim = Sir(init_infect, filename, delimiter)
    sim.draw_network()
    sim.simulation(p_trans, t_rec, t_sus, t_sim)
    sim.draw_network()

```

La funzione prende come input tutti i parametri necessari all'inizializzazione di un oggetto **Sir**: **nodi infetti iniziali**, **file csv**, **p_trans**, **t_rec**, **t_sus**, **t_sim** e **num_sim** (**numero di simulazioni**).

Il numero di simulazioni permette di applicare due tipi di simulazioni differenti: se vogliamo eseguire una sola simulazione, questa verrà riportata in modo completo proiettando in output anche le visualizzazioni grafiche della rete.

Invece con **num_sim > 1** si eseguono una serie di simulazioni: si andranno a creare tot reti, simulate e per ognuna di essa vengono determinati i nodi super-spreaders.

Tutti i risultati delle simulazioni vengono salvati in un dataframe pandas che verrà infine filtrato e classificato per le maggiori occorrenze dei nodi super-spreaders presenti in tutte le simulazioni.

4 Simulazione epidemica

Per verificare ed analizzare il funzionamento del sistema implementato, ho deciso di utilizzare due big network prese direttamente da <https://snap.stanford.edu/data/>:

- **ego-Facebook:**
 - `facebook_combined.csv` (4039 nodi)
- **twitch-gamers:**
 - `musae_ENGB.edges.csv` (6960 nodi)

La prima rete può essere considerata una "base" large network, in cui in numero di 4000 nodi permettono di distinguersi ed essere ritenuta una rete grande, ma non di grande complessità; mentre la seconda rete, con quasi 7000 nodi, risulta una large network con un importante dimensione.

Date queste due largest network, si vanno ad effettuare delle simulazioni random andando a modificare i vari parametri di input.

Le simulazioni vengono avviate richiamando il metodo **random_simulation**.

Ad esempio:

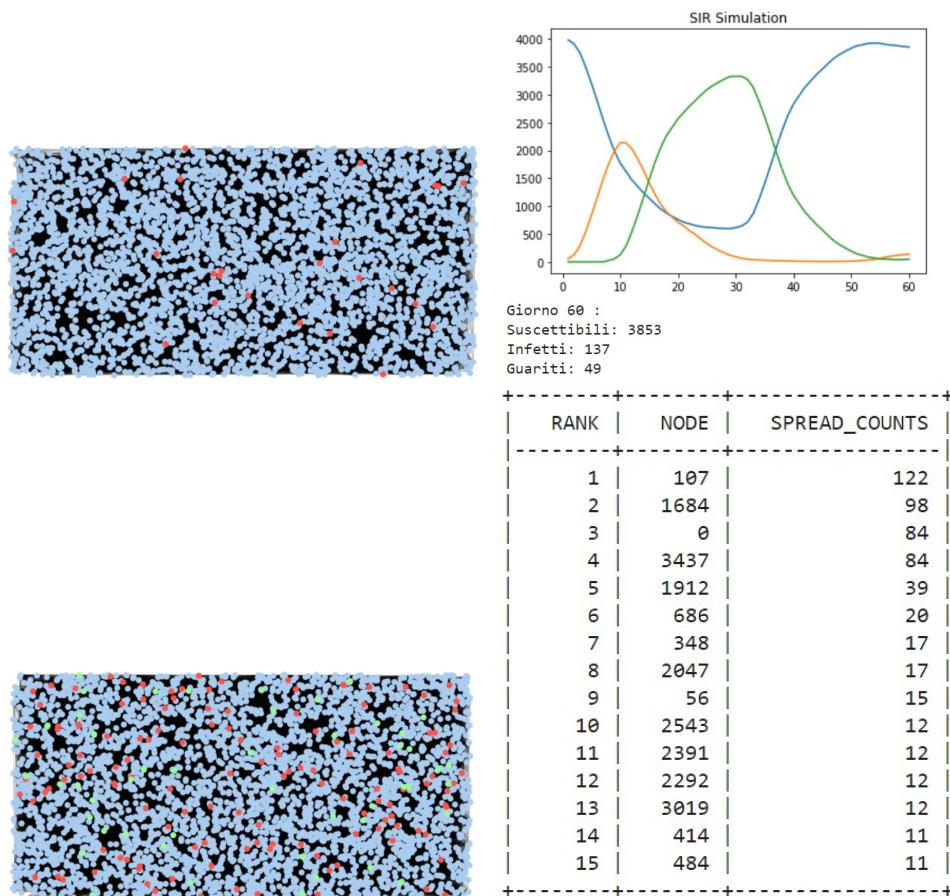
```
#Esecuzione di una singola simulazione
random_simulation(25, 'facebook_combined.csv', ' ', 0.02, 7, 21, 60, 1)

#Esecuzione di una serie di simulazioni (num_sim > 1)
random_simulation(25, 'facebook_combined.csv', ' ', 0.02, 7, 21, 60,
50)
```

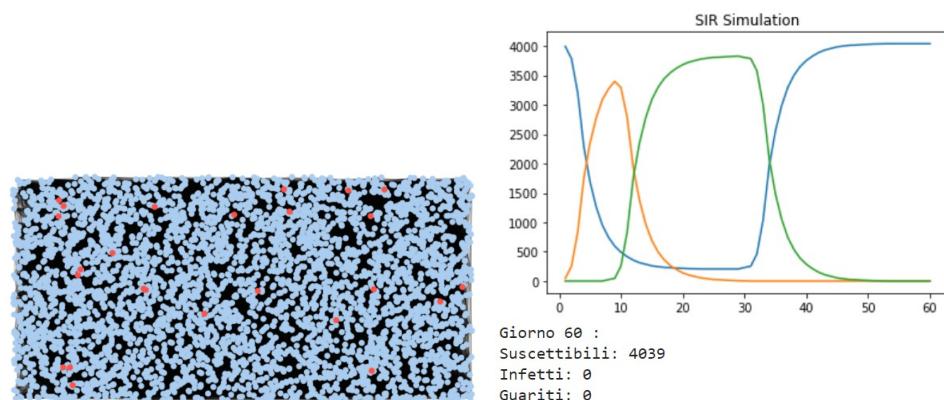
4.1 Simulazioni singole

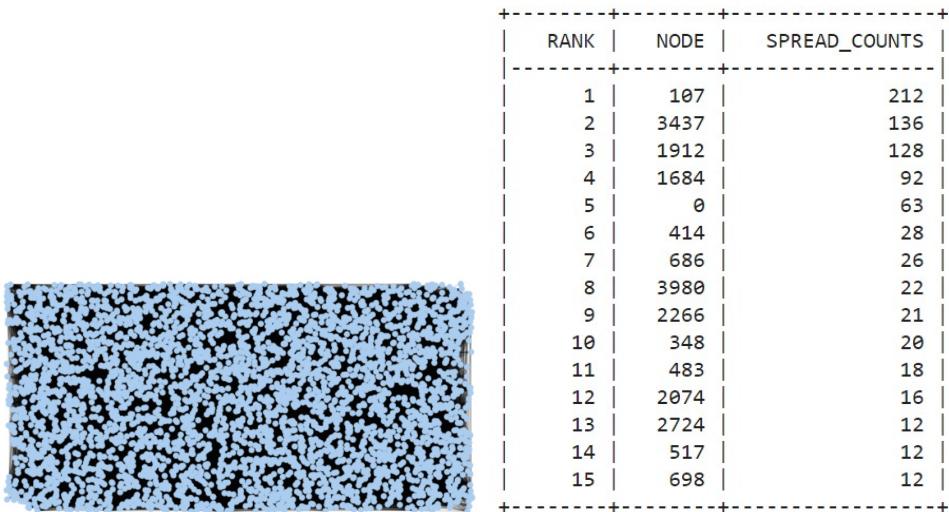
Risultati di alcune simulazioni con l'utilizzo delle due network:

```
#Simulazione con probabilita trasmissione al 2%
#input parametri simulazione:p_trans, t_rec, t_sus, t_sim, num_sim
random_simulation(25, 'facebook_combined.csv', ' ', 0.02, 7, 21, 60, 1)
```



```
#Seconda simulazione con probabilit infezione al 5%
#input parametri simulazione:p_trans, t_rec, t_sus, t_sim, num_sim
random_simulation(25, 'facebook_combined.csv', ' ', 0.05, 7, 21, 60, 1)
```



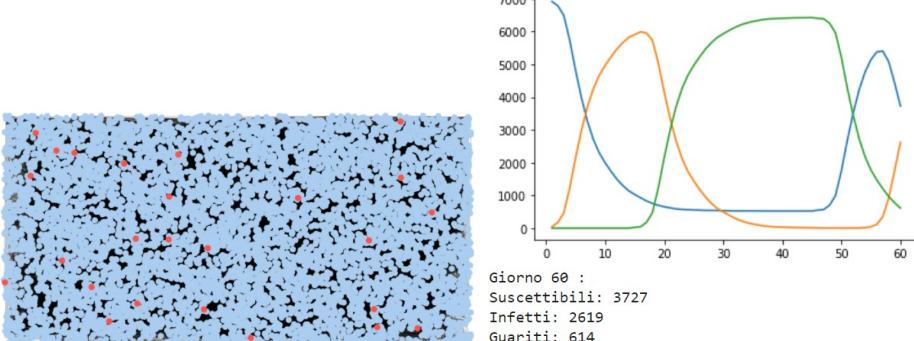


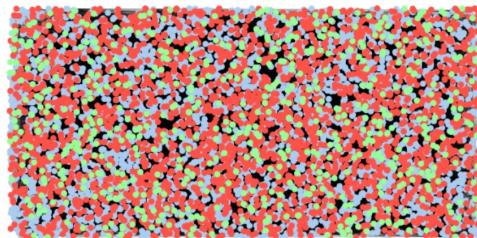
Le due simulazioni avvengono con lo stesso parametro iniziale di nodi infetti, ma quest'ultimi sono assegnati sempre randomicamente.

L'andamento dell'epidemia è condizionata da i parametri iniziali della simulazione e dalla "influenza" dei nodi inizialmente infetti.

Quello che si può più notare è che con l'aumentare della probabilità di trasmissione (`p_trans`) il picco degli infetti nei primi tempi iniziali è maggiore, quindi si ha inizialmente un maggiore spread di infezione.

```
#Simulazione con probabilità di infezione al 7%
#input parametri simulazione:p_trans, t_rec, t_sus, t_sim, num_sim
random_simulation(25, 'musae_ENGB.edges.csv', ',', 0.07, 14, 30, 60, 1)
```

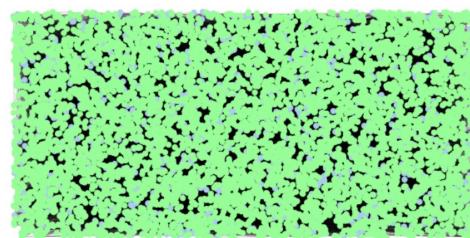
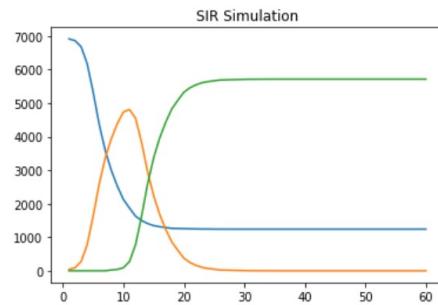
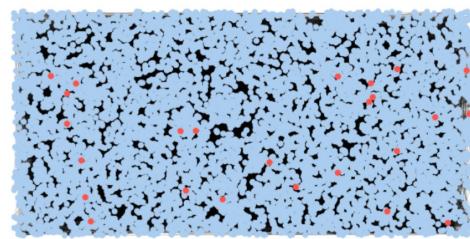




RANK	NODE	SPREAD_COUNTS
1	4949	374
2	1773	274
3	3401	172
4	166	131
5	1103	121
6	5842	116
7	6136	108
8	581	104
9	2447	88
10	1924	86
11	2732	85
12	4406	63
13	1598	63
14	4639	56
15	3706	53

```
#Simulazione con t_sus -> infinito (999)
```

```
#input parametri simulazione:p_trans, t_rec, t_sus, t_sim, num_sim
random_simulation(25, 'musae_ENGB_edges.csv', ',', 0.07, 7, 999, 60, 1)
```



RANK	NODE	SPREAD_COUNTS
1	4949	177
2	1773	137
3	3401	101
4	5842	85
5	6136	59
6	166	56
7	1924	55
8	581	53
9	2447	50
10	1598	48
11	3706	48
12	1103	46
13	5269	46
14	4016	42
15	5945	41

Con questo caso particolare, avremo una guarigione completa con immunità. I nodi guariti non saranno più vulnerabili alla malattia; i nodi infetti di conseguenza, non potendo trasmettere la malattia, tenderanno alla scomparsa.

L'individuazione dei nodi super-spreaders può essere effettuata anche analizzando una singola simulazione. Naturalmente osservando soltanto una sola simulazione, i risultati potrebbero essere a volte imprecisi.

4.2 Random Simulation

Eseguendo una serie di simulazioni random si possono individuare con maggiore accuratezza i nodi super-spreaders durante le simulazioni.

Per le entrambi rete utilizzate, si sono effettuate 200 simulazioni su ciascuna rete: per ogni singola simulazione vengono determinati i nodi super spreaders; i migliori nodi che compaiono nel maggior numero di simulazioni saranno i nodi super spreaders finali.

```
#Random simulation with Facebook Network
random_simulation(25, 'facebook_combined.csv', ',', 0.05, 7, 21, 60,
300)

#Random simulation with Twiith ENGB network
random_simulation(25, 'musae_ENGB_edges.csv', ',', 0.05, 7, 21, 60,
300)
```

Risultati per Facebook Network:

RANK	NODE	FREQUENCY
1	0	300
2	3437	300
3	1912	300
4	1684	300
5	107	300
6	348	295
7	414	283
8	686	274
9	483	151
10	698	130
11	3980	117
12	3830	55
13	2543	52
14	376	51
15	2347	45

Risultati per Twitch ENGB Network:

RANK	NODE	FREQUENCY
1	1773	300
2	166	300
3	6136	300
4	4949	300
5	3401	300
6	1924	300
7	5842	299
8	581	294
9	2732	291
10	1103	287
11	2447	283
12	4016	209
13	3706	207
14	4196	178
15	2481	137

Infine si può anche confrontare tali risultati gli algoritmi di calcolo delle centralità dei nodi: si controlla se parte dei nodi super-spreaders sono anche nodi che hanno alto valore di centralità per la rete.

Facebook e Twitch Centrality Metrics

	BC	CC	EC	DC	VR		BC	CC	EC	DC	VR
1	107	107	1912	107	107	1	1773	4949	4949	1773	1773
2	1684	58	2266	1684	1684	2	4949	1773	1773	4949	4949
3	3437	428	2206	1912	1912	3	3401	3401	3401	3401	3401
4	1912	563	2233	3437	3437	4	5842	5842	5842	6136	6136
5	1085	1684	2464	0	0	5	166	2447	1924	166	166
6	0	171	2142	2543	2543	6	6136	1924	2447	5842	5842
7	698	348	2218	2347	2347	7	1924	792	6136	1924	1924
8	567	483	2078	1888	1888	8	581	2997	4196	581	581
9	58	414	2123	1800	1800	9	2447	6716	792	2732	2732
10	428	376	1993	1663	348	10	2732	4016	6716	2447	2447
11	563	475	2410	2266	483	11	4016	2345	4016	1103	1103
12	860	566	2244	1352	2266	12	1103	1089	2732	4196	4196
13	414	1666	2507	483	1663	13	3706	581	581	4016	4016
14	1577	1534	2240	348	1352	14	4196	3285	2345	3706	3706
15	348	484	2340	1730	1941	15	2997	1869	1103	2481	2481

Per entrambi le reti si può notare che fra i nodi super-spreaders più importanti, come i migliori 5, sono quei nodi che hanno anche un altissimo valore di centralità. Ciò significa che questi sono anche nodi "hub" della rete, quindi possono con moltissima facilità trasmettere la malattia un tutta la rete.

Per gli altri nodi, invece, si nota una sempre un buon livello di centralità per le diverse misure, ciò nonostante questi nodi non sono sempre presenti costantemente in tutte le simulazioni; quindi anche avendo un'importanza di essere un "hub" per la struttura della rete, ciò non porta che siano anche degli ottimi e costanti trasmettitori.