Majority Cascade in Cost Networks

Corso di **Reti Sociali**, a.a. 2023-2024
proff. Luisa Gargano, Adele Anna Rescigno

Antonio Gravino (matr. 0522501502)
Dario Trinchese (matr. 0522501493)
Carmine Napolitano (matr. 0522501538)

GitHub Repository: /cost-majority-cascade/

Indice dei contenuti

1 Dataset utilizzato

1.1 Descrizione del dataset

Il dataset scelto per il progetto è *egoFacebook*, disponibile su **SNAP** (Stanford Network Analysis Platform): una libreria attiva dal 2004 che tutt'oggi è in crescita organica grazie all'attività di ricerca nell'analisi di grandi reti sociali e informative. In generale, la libreria offre dataset di reti di vario genere: dai dati relativi a Facebook, Twitter, fino a Wikipedia.

Questo dataset consiste in 'circles' (o 'liste di amici') provenienti da diversi utenti Facebook. I dati sono stati prelevati utilizzando un'applicazione di Facebook ad oggi non più disponibile a causa di un cambio di politiche di Meta, azienda proprietaria del social network. Dataset simili attualmente possono essere generati utilizzando l'API Graph di Facebook, il metodo principale tramite cui le app possono leggere e scrivere nel social graph di Facebook. È possibile ottenere l'accesso all'API Graph di Facebook creando un'applicazione sulla sezione *Facebook for Developers* di Facebook ed ottenere una API Key, da utilizzare poi nell'applicazione per inviare richieste all'API Graph tramite HTTP GET o POST.

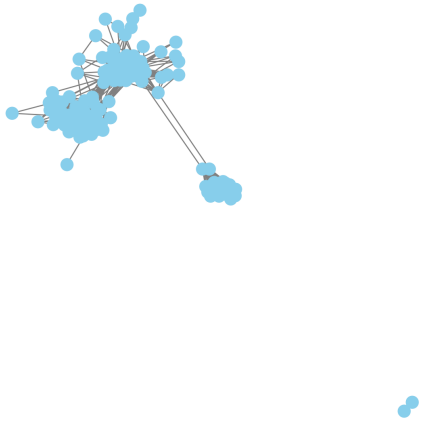
1.2 Caratteristiche del dataset

Il dataset utilizzato presenta al proprio intero 4 reti, ognuna rappresentante la lista di amici di uno specifico utente Facebook che, per motivi di privacy, è reso anonimo. Le reti presentano una struttura abbastanza simile, tutte presentano una componente gigante che comprende la maggior parte dei nodi (per ogni rete, la componente gigante contiene almeno il 97% dei nodi.). Le reti differiscono nel numero dei nodi; la rete più piccola contiene appena 150 nodi, la più grande contiene 1034 nodi. La rete più piccola, rappresentante la rete di amicizie dell'individuo anonimo 414, contiene la percentuale più alta di triangoli. La percentuale di triangoli è stata calcolata come il rapporto tra i triangoli presenti nel grafo e i possibili triangoli del grafo $\binom{|V|}{3}$. Seguono ulteriori dettagli sulle reti nella seguente tabella:

Rete	**Nodi**	**Edges**	**Size CG**	**CG (%)**	**Triangoli**	**Tr. (%)**
Rete 414	150	1693	148	98.67%	10618	1.93%
Rete 348	224	3192	224	100%	23503	1.27%
Rete 0	333	2519	324	97.30%	10740	0.18%
Rete 107	1034	26749	1034	100%	420329	0.23%

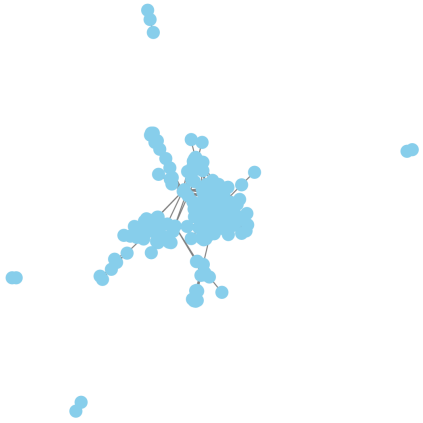
Table 1: Dati delle reti nel dataset

1.3 Visualizzazione del dataset

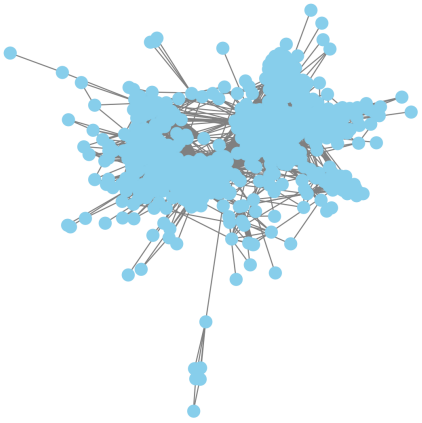


(a) Rete 414

(b) Rete 348



(c) Rete 0



(d) Rete 107

Le immagini sono state generate con la libreria **matplotlib** di Python. Per favorire la visualizzazione è stato usato come algoritmo di posizionamento dei nodi lo Spring Layout della libreria **networkx**. Le reti presentano pressochè strutture simili anche graficamente. Dove la componente gigante non comprende la totalità dei nodi è possibili osservare pochi nodi sparsi ed isolati, lontani dal centro.

1.4 Scelta della rete

Per le nostre sperimentazioni è stata scelta la **rete 348**, il miglior compromesso tra dimensione e complessità; la scelta di questa specifica rete è giustificata dai tempi di esecuzione *accettabili* per l'ottenimento dei risultati finali da valutare e commentare relativi alle nostre sperimentazioni.

2 Implementazione

La rete scelta è stata implementata in Python sfruttando NetworkX, un package per la creazione e manipolazione delle dinamiche e delle funzioni di reti complesse.

In particolare, ogni algoritmo implementato sfrutta i seguenti import:

```
1 import math
2 import networkx
3 from networkx import Graph
4 from collections import Counter
```

2.1 Algoritmo Costs-Seeds-Greedy

L'idea alla base di questo algoritmo è quello di andare ad inserire nel seedset il nodo che permette di massimizzare l'influenza ma non solo, infatti è di cruciale importanza anche il costo del nodo stesso poiché, siccome si vuole un seedset massimale il nodo scelto deve sia massimizzare l'influenza ma allo stesso tempo costare poco. Il nodo selezionato è quindi quello che ad ogni iterazione finché $c(S) < k$ va a massimizzare il rapporto sequente 1:

$$\frac{\Delta_v f_i\left(S_d\right)}{c(u)}, \tag{1}$$

dove $\Delta_i f_i(S_d) = f(S_d \cup u - f(S_d))$, mentre $c(u)$ è il costo associato al nodo. Quindi il rapporto risulta essere inversamente proporzionale al costo stesso.

L'algoritmo Costs-Seeds-Greedy è stato implementato con tre diverse funzioni submodulari f_i:

- $f_1(S)$: $\sum_{v \in V} \min\left\{|N(v) \cap S|, \left\lceil \frac{d(v)}{2} \right\rceil\right\}$
- $f_2(S)$: $\sum_{v \in V} \sum_{i=1}^{|N(v) \cap S|} \max\left\{\left\lceil \frac{d(v)}{2} \right\rceil - i + 1, 0\right\}$
- $f_3(S)$: $\sum_{v \in V} \sum_{i=1}^{|N(v) \cap S|} \max\left\{\frac{\left\lceil \frac{d(v)}{2} \right\rceil - i + 1}{d(v) - i + 1}, 0\right\}$

L'implementazione relativa viene mostrata nella sezione successiva con le funzioni *first*, *secondo* e *third*.

```
1 def cost_seeds_greedy(graph: Graph, threshold: int, cost_function:
  ↪ callable, submodular_function: callable, verbose: bool):
2     Sd = []
```

```
    Sp = []
    step = 0

    while cost_function(graph, Sd) <= threshold:
        u = argmaxselect(graph, submodular_function, Sd, cost_function,
        ↪  verbose)
        Sp = Sd.copy()
        Sd.append(u)
        step = step + 1

    return Sp
```

```
def argmaxselect(G, submodular_function, Sd, cost_function, verbose):
    tempSet = list(G.nodes())
    if verbose:
        print("\t[ARGMAXSELECT] : V = ", tempSet)
        print("\t[ARGMAXSELECT] : Sd = ", Sd)
    set = tempSet.copy()
    for u in tempSet:
        if u in Sd:
            set.remove(u)
    if verbose:
        print("\t[ARGMAXSELECT] : V-Sd = ", set)
    if not set:
        if verbose:
            print("\t[ARGMAXSELECT] : V-Sd is empty!")
        return None  # Restituisci None se set è vuoto

    bestNode = set[0]
    bestValue = 0.0
    for v in set:
        if verbose:
            print("\n\t[ARGMAXSELECT] : f(Sd U ",v,") = ",
            ↪  submodular_function(G, Sd + [v]))
            print("\t[ARGMAXSELECT] : f(Sd) = ", submodular_function(G,
            ↪  Sd))
            print("\t[ARGMAXSELECT] : c(",v,") = ", cost_function(G,v))

        value = (submodular_function(G, Sd + [v]) -
        ↪  submodular_function(G, Sd)) / cost_function(G,v)

        if verbose:
            print("\t[ARGMAXSELECT] : value = ", value)
        if value>bestValue:
```

```
30              bestValue = value
31              bestNode = v
32      return bestNode
```

2.1.1 Funzioni submodulari

Il primo algoritmo sfrutta delle funzioni submodulari. Vengono presentate tre possibili funzioni submodulari, di cui l'ultima di nostra ideazione.

```
1  def first(G, s):
2      sum = 0
3      for v in list(G.nodes()):
4          first_term = len(list(set(list(G.neighbors(v))) & set(s)))
5          second_term = math.ceil(G.degree(v)/2)
6          sum += min(first_term,second_term)
7      return sum
```

```
1  def second(G, s):
2      sum = 0
3      for v in list(G.nodes()):
4          for i in range(1, len(list(set(list(G.neighbors(v))) & set(s)))
           ↪  + 1):
5              first_term = math.ceil(G.degree(v)/2) - i + 1
6              second_term = 0
7              sum += max(first_term,second_term)
8      return sum
```

```
1  def third(G, s):
2      sum = 0
3      for v in list(G.nodes()):
4          for i in range(1, len(list(set(list(G.neighbors(v))) & set(s)))
           ↪  + 1):
5              first_term = (math.ceil(G.degree(v)/2) - i + 1)/(G.degree(v)
               ↪  - i + 1)
6              second_term = 0
7              sum += max(first_term,second_term)
8      return sum
```

2.2 Algoritmo Weighted Target Set Selection (WTSS)

Il WTSS è uno dei tre algoritmi utilizzati per selezionare il seed set che mi permetta di influenzare la maggior parte dei nodi nella reti. La condizione è che il $c(S) < k$, cioè che la somma dei costi dei nodi nel *seedset* sia inferiore al *budget* fissato.

```
def WTSS(G,k, cost_function: callable):
    G_copy = G.copy()
    assign_treshold(G_copy)
    costs(G_copy, cost_function)

    seed_set = []
    to_delete = None
    third_case_node_selection = Counter()

    sum = 0

    while len(G_copy.nodes()) > 0:

        first_case_activated = False

        for node in G_copy.nodes():
            if G_copy.nodes[node]["t"] == 0:
                first_case_activated = True
                to_delete = node
                neighbors = list(G_copy.neighbors(node))
                for neighbor in neighbors:
                    treshold = G_copy.nodes[neighbor]["t"]
                    G_copy.nodes[neighbor]["t"] = max(0,(treshold - 1))
                G_copy.remove_node(to_delete)
                break

        if first_case_activated:
            continue

        second_case_activated = False

        for node in G_copy.nodes():
            if G_copy.degree(node) < G_copy.nodes[node]["t"]:
                second_case_activated = True
                seed_set.append(node)
                sum += G_copy.nodes[node]["cost"]
                to_delete = node
                neighbors = list(G_copy.neighbors(node))
                for neighbor in neighbors:
```

```
                    treshold = G_copy.nodes[neighbor]["t"]
                    G_copy.nodes[neighbor]["t"] -= 1
                G_copy.remove_node(to_delete)
                break

        if second_case_activated:
            continue

        third_case_node_selection = Counter()
        for node in G_copy.nodes():
            cost = G_copy.nodes[node]["cost"]
            threshold = G_copy.nodes[node]["t"]
            degree = G_copy.degree(node)
            third_case_node_selection[node] = ((cost * threshold) /
            ↪  (degree * (degree + 1)))
        to_delete = max(third_case_node_selection,
        ↪  key=third_case_node_selection.get)
        G_copy.remove_node(to_delete)

        if sum > k:
            return seed_set[:-1]
    return seed_set
```

```
def assign_treshold(G):
    for node in G.nodes():
        G.nodes[node]["t"] = math.ceil(G.degree(node)/2)

def costs(G, cost_function: callable):
    for node in G.nodes():
        G.nodes[node]["cost"] = cost_function(G,node)

def print_costs(G):
    for node in G.nodes():
        print("Costo del nodo",node,":",G.nodes[node]["cost"])

def print_tresholds(G):
    for node in G.nodes():
        print("Treshold del nodo",node,":",G.nodes[node]["t"])
```

Si inizia con il fare una copia del grafo considerato, per far si che le modifiche apportate non vadano a modificare il grafo originario, evitando cosi di dover caricare il grafo ad ogni esecuzione di WTSS. Sul grafo considerato andiamo poi a settare il threshold ed il costo associato ad ogni nodo. Il processo di riempimento del seedset è iterativo, e si continua finché $c(S) < k$. Chiaramente anche il caso in cui non ci sono più nodi nel

grado reppresenta una condizione di uscita, seppur poco probabile. Ciò che viene fatto dall'algoritmo WTSS è riassumibile nei seguenti passi:

- **Passo 1**: Se esiste un nodo v nel nel grafo la cui soglia corrente $k(v) = 0$ allora lo seleziono come nodo da eliminare e per tutti i vicini u di v vado ad impostare la soglia come $k(u) = max(0, k(u) - 1)$. Se non trovo un nodo che soddifa questa condizione vado al passo 2.
- **Passo 2**: Se esiste un nodo v nel nel grafo il cui grado corrente è minore della soglia corrente $\delta(v) < k(v)$ allora lo mettiamo nel seedset poiché rappresenta un nodo che non può essere attivato. A tutti i vicini u di v andiamo a diminuire la soglia di 1. Se non trovo un nodo che soddisfo questa condizione vado al passo 3.
- **Passo 3**: Se arrivo al passo 3 vado a selezionare il nodo che massimizza il rapporto definito nell'equazione 2:

$$v = \text{argmax}_{u \in U} \left\{ \frac{c(u)k(u)}{\delta(u)(\delta(u)+1)} \right\}, \tag{2}$$

  dove $c(u)$, $k(u)$, $\delta(u)$ sono ripettivamente il costo, la soglia ed il grado associato al nodo u. L'idea è quella di andare ad eliminare il nodo che ha meno probabilità di attivarsi. Finito il passo 3, si ritorna al passo 1 finché una delle condizioni di uscita, sopraccitate viene raggiunta.

2.3 Algoritmo My-Seeds

Il seguente algoritmo cerca di selezionare iterativamente il nodo che contribuirà maggiormente all'aumento della copertura totale della rete. Come nel WTSS anche qui andiamo a lavorare su una copia del grafo per le medesime ragione. In questo caso però il nodo che viene selezionato da mettere nel seedset è quello che ha uno *score* maggiore, dove lo score di un nodo v è definito come il prodotto tra il coefficente di clustering del nodo e il grado dello stesso rapportato al costo del nodo. Quindi in linea generale, si va a stimare l'importanza del nodo sfruttando il coefficente di clustering, ma come detto in precedenza anche il costo del nodo è determinante nella scelta poiché si vuole un seedset massimale. Il coefficente di clustering è una misura che quantifica quanto i vicini di un nodo in una rete sono collegati tra loro. Esso viene utilizzato per valutare quanto una rete sociale sia "clusterizzata" o formi gruppi densamente interconnessi. Esistono diverse definizioni di coefficienti di clustering, ma una delle più comuni riguarda il coefficiente di clustering locale (o coefficiente di clustering di un nodo), che è calcolato come il rapporto tra il numero di collegamenti effettivi tra i vicini di un nodo e il numero massimo possibile di collegamenti tra di loro, quindi è un rapporto tra i casi favorevoli e quelli possibili. Ciò che è stato appena descritto può essere sintetizzato dall'equazione 3:

$$C_i = \frac{L_i}{\binom{k_i}{2}} = \frac{2L_i}{k_i\,(k_i - 1)}, \tag{3}$$

dove C_i rappresenta il coefficiente di clustering del nodo i, L_i corrisponde al numero di edges esistenti tra i vicini del nodo i e k_i equivale al grado del nodo i.

```python
def my_seeds(G: Graph, k: int, cost_function: callable) -> list:
    costs(G, cost_function)

    sum_costs = 0
    remaining_graph = G.copy()
    seed_set = []

    while sum_costs <= k:
        selected_node = None
        best_score = 0

        for node in remaining_graph.nodes():
            score = networkx.clustering(G, node) * G.degree(node) /
             ↪  remaining_graph.nodes[node]["cost"]

            if score > best_score:
                best_score = score
                selected_node = node

        if selected_node == None:
            break

        sum_costs += remaining_graph.nodes[node]["cost"]
        seed_set.append(selected_node)
        remaining_graph.remove_node(selected_node)

    return seed_set
```

2.4 Classe SpreadingAlgorithm

Viene definita la classe `SpreadingAlgorithm`, la quale si occupa di gestire l'algoritmo di diffusione all'interno del grafo. Durante l'inizializzazione, vengono passati diversi parametri fondamentali, tra cui l'indice dell'algoritmo selezionato, l'indice della funzione submodulare selezionata, il grafo stesso, la soglia di diffusione, la funzione di costo e un flag per abilitare la modalità verbosa.

Una volta inizializzata, la classe configura la funzione submodulare in base all'indice specificato, qualora venga scelto il primo algoritmo fra i tre disponbiili. Questo viene fatto attraverso il metodo `setup_submodular_function`, che associa l'indice selezionato a una specifica funzione submodulare definita nel modulo `helpers.submodular_functions`.

Successivamente, il metodo `get_seed_set` determina il seed set ottimale in base all'algoritmo selezionato e alla funzione di costo. Se l'algoritmo è `COST_SEEDS_GREEDY`, viene invocata la funzione `cost_seeds_greedy` del modulo `algorithms.cost_seeds_greedy`, passando il grafo, la soglia di diffusione, la funzione di costo e la funzione submodulare. Altrimenti, se l'algoritmo è `WTSS` o `MY_SEEDS`, vengono chiamate le rispettive funzioni `WTSS` o `my_seeds` dei moduli `algorithms.WTSS` e `algorithms.my_seeds`. Infine, il seed set ottenuto viene restituito.

```
1  import algorithms.cost_seeds_greedy
2  import algorithms.WTSS
3  import algorithms.my_seeds
4  import helpers.submodular_functions as sf
5
6  from default_config import Algorithms
7
8  class SpreadingAlgorithm():
9      def __init__(self, selected_algorithm_index,
       ↪  selected_submodular_function_index, graph, threshold,
       ↪  cost_function, verbose):
10         self.selected_algorithm_index = selected_algorithm_index
11         self.selected_submodular_function_index =
           ↪  selected_submodular_function_index
12
13         self.verbose = verbose
14
15         self.G = graph
16         self.threshold = threshold
17         self.cost_function = cost_function
18         self.submodular_function = self.setup_submodular_function()
19
20
21     def setup_submodular_function(self):
22         if self.selected_submodular_function_index == 1:
23             submodular_function = sf.first
24         elif self.selected_submodular_function_index == 2:
25             submodular_function = sf.second
26         elif self.selected_submodular_function_index == 3:
27             submodular_function = sf.third
28         else:
29             submodular_function = None
30
31         return submodular_function
32
33
34     def get_seed_set(self):
```

```
        if self.selected_algorithm_index ==
        ↪  Algorithms.COST_SEEDS_GREEDY.value:
            seed_set = algorithms.cost_seeds_greedy.cost_seeds_greedy(
                graph = self.G,
                threshold = self.threshold,
                cost_function = self.cost_function,
                submodular_function = self.submodular_function,
                verbose = self.verbose
            )
        elif self.selected_algorithm_index == Algorithms.WTSS.value:
            seed_set = algorithms.WTSS.WTSS(
                G = self.G,
                k = self.threshold,
                cost_function = self.cost_function
            )
        elif self.selected_algorithm_index == Algorithms.MY_SEEDS.value:
            seed_set = algorithms.my_seeds.my_seeds(
                G = self.G,
                k = self.threshold,
                cost_function = self.cost_function
            )

        return seed_set
```

```
def costs(G: Graph, cost_function: callable) -> None:
    for node in G.nodes():
        G.nodes[node]["cost"] = cost_function(G,node)


def total_cost(G: Graph) -> int:
    total_cost = 0

    for node in G.nodes():
        total_cost += node["cost"]
```

2.5 Funzioni di costo e classe CostFunctionFactory

Le funzioni `first`, `second` e `third` sono definite per calcolare i costi associati ai nodi o ai sottoinsiemi di nodi all'interno del grafo. Queste funzioni vengono utilizzate all'interno del primo step (individuazione seed set) del processo di diffusione per determinare il costo di selezionare determinati nodi come parte del seed set.

La funzione `first` calcola il costo sommando i valori associati ai nodi in un certo vettore casuale ("cost map"). La funzione `second` utilizza il grado dei nodi nel grafo per calcolare il costo, utilizzando la funzione `math.ceil` per arrotondare verso l'alto il risultato. Infine, la funzione `third` calcola il costo basato sul quadrato del grado dei nodi nel grafo, normalizzato rispetto al massimo grado presente nel grafo.

```
def first(self, graph: Graph, v):
        if isinstance(v, list):
            costSum = 0

            for u in v:
                costSum += self.cost_map[u]
            return costSum
        else:
            return self.cost_map[v]
```

```
def second(self, G: Graph, v):
        if isinstance(v, list):
            costSum = 0

            for u in v:
                costSum += math.ceil(G.degree(u) / 2)
            return costSum
        else:
            return math.ceil(G.degree(v) / 2)
```

```
def third(self, G: Graph, v):
        if isinstance(v, list):
            costSum = 0

            for u in v:
                costSum += math.ceil((G.degree(u) ** 2) / self.d_max)
            return costSum
        else:
            return math.ceil((G.degree(v) ** 2) / self.d_max)
```

La classe `CostFunctionFactory` è progettata per generare funzioni di costo utilizzate nel processo di diffusione all'interno di un grafo, in maniera programmatica. Durante l'inizializzazione, vengono passati diversi parametri essenziali, tra cui l'indice dell'algoritmo selezionato, il grafo stesso, il range minimo e massimo dei costi (relativi alla prima funzione di costo, che prevede una cost map casuale), il grado massimo del grafo e un flag per abilitare la modalità verbosa.

Il metodo `get_function` della classe determina la funzione di costo appropriata in base

all'indice dell'algoritmo selezionato. Se l'indice non è compreso tra 1 e 3, viene sollevata un'eccezione. Altrimenti, l'indice viene utilizzato per associare il giusto metodo di calcolo del costo (`first`, `second` o `third`). Infine, la funzione corrispondente viene restituita per essere utilizzata nel processo di diffusione.

```
class CostFunctionFactory():
    def __init__(self, selected_algorithm_index, graph, range_min,
    ↪  range_max, d_max, verbose = False):
        self.selected_algorithm_index = selected_algorithm_index

        self.G = graph
        self.range_min = range_min
        self.range_max = range_max
        self.d_max = d_max

        self.verbose = verbose

        self.cost_map = self.create_cost_map()


    def get_function(self):
        if self.selected_algorithm_index not in [1, 2, 3]:
            raise
            ↪  IndexError("La funzione di costo indicata non è valida.")

        if self.selected_algorithm_index == 1:
            fn = self.first
        elif self.selected_algorithm_index == 2:
            fn = self.second
        elif self.selected_algorithm_index == 3:
            fn = self.third

        return fn
```

2.6 Processo di diffusione e classe CascadeProcess

La classe `CascadeProcess` è progettata per simulare il processo di diffusione all'interno di un grafo di rete sociale partendo da un insieme di nodi iniziali, noto come seed set. Durante l'inizializzazione, vengono passati il grafo, il seed set e un flag per abilitare la modalità verbosa.

Il metodo `get_influenced_nodes` esegue la simulazione del processo di diffusione. Viene utilizzato un approccio iterativo, in cui si aggiungono nodi all'insieme influenzato finché

non si raggiunge la stabilità, cioè finché l'insieme influenzato non si modifica più in una nuova iterazione. All'interno del ciclo while, vengono confrontati gli insiemi di nodi influenzati nelle iterazioni precedente e attuale per determinare se sono stati influenzati nuovi nodi. Questo processo continua finché non viene raggiunta la stabilità dell'insieme influenzato, e infine viene restituito l'insieme completo dei nodi influenzati.

```
class CascadeProcess():
    def __init__(self, graph: Graph, seed_set: list, verbose: bool) ->
     ↪  None:
        self.graph: Graph = graph
        self.seed_set: list = seed_set
        self.verbose: bool = verbose


    def get_influenced_nodes(self) -> list:
        prev_influenced: list = []
        influencing: list = copy.deepcopy(self.seed_set)

        while len(influencing) != len(prev_influenced):

            prev_influenced: list = copy.deepcopy(influencing)
            casted_prev_influenced = [str(node) for node in
             ↪  prev_influenced]

            for node in self.graph.nodes():
                if node in casted_prev_influenced:
                    continue

                intersection = len([x for x in casted_prev_influenced if
                 ↪  x in list(self.graph.neighbors(node))])

                degree_threshold = math.ceil(self.graph.degree(node) /
                 ↪  2)

                if intersection >= degree_threshold:
                    influencing.append(int(node))

        return influencing
```

2.7 Caricamento della rete e classe NetworkLoader

La funzione `read_network` è progettata per leggere e creare un grafo da due file di testo, uno contenente gli archi (`.EDGES`) e l'altro contenente i nodi (`.CIRCLES`). All'interno

di questa funzione, sono definite tre sotto-funzioni: `read_edges`, `read_circles` e `create_graph`. La prima legge gli archi dal file `.EDGES`, la seconda legge i cerchi dal file `.CIRCLES`, e la terza crea il grafo incorporando gli archi e associando i cerchi come attributi dei nodi. Una volta letti i file e creato il grafo, la funzione restituisce il grafo stesso.

La funzione `main` costituisce l'entry point del programma. All'interno di questa funzione, vengono definiti i percorsi ai file `.EDGES` e `.CIRCLES`, e successivamente viene chiamata la funzione `read_network` per leggere il network dal file e ottenere il grafo. Viene quindi stampato un riepilogo delle informazioni del grafo utilizzando `nx.info`, e infine il grafo viene visualizzato utilizzando `nx.draw`. Se lo script viene eseguito direttamente, la funzione `main` viene chiamata per eseguire il programma.

```
def read_network(edges_file_path, circles_file_path):
    # Funzione per leggere i file .EDGES
    def read_edges(file_path):
        edges = []
        with open(file_path, 'r') as file:
            for line in file:
                nodes = line.strip().split()
                if len(nodes) == 2:
                    edges.append((nodes[0], nodes[1]))
        return edges

    # Funzione per leggere i file .CIRCLES
    def read_circles(file_path):
        circles = {}
        with open(file_path, 'r') as file:
            for line in file:
                parts = line.strip().split()
                if len(parts) > 1:
                    circle_id = parts[0]
                    members = parts[1:]
                    circles[circle_id] = members
        return circles

    # Funzione per creare il grafo
    def create_graph(edges, circles):
        G = nx.Graph()

        # Aggiungi gli archi al grafo
        G.add_edges_from(edges)

        # Aggiungi i cerchi come attributi dei nodi
        for circle_id, members in circles.items():
```

```
            for member in members:
                if member in G.nodes:
                    if 'circles' not in G.nodes[member]:
                        G.nodes[member]['circles'] = []
                    G.nodes[member]['circles'].append(circle_id)

        return G

    # Leggi i file
    edges = read_edges(edges_file_path)
    circles = read_circles(circles_file_path)

    # Crea il grafo
    G = create_graph(edges, circles)

    return G

def main():
    # Percorsi ai file .EDGES e .CIRCLES
    edges_file_path = 'sample_networks/0.edges'
    circles_file_path = 'sample_networks/0.circles'

    # Leggi il network
    G = read_network(edges_file_path, circles_file_path)

    # Stampa informazioni sul grafo
    print(nx.info(G))

    # Disegna il grafo
    pos = nx.spring_layout(G)
    nx.draw(G, pos, with_labels=True)
    plt.show()

if __name__ == "__main__":
    main()
```

2.8 Networking Mocking con Erdos-Reyni

Il codice fornito genera un grafo casuale utilizzando il modello di Erdős-Rényi con una probabilità p di collegamento tra i nodi. Il numero totale di nodi nel grafo è n. Il grafo viene quindi salvato in due file di testo: uno contenente gli archi del grafo con estensione `.EDGES`, e l'altro contenente i nodi con estensione `.CIRCLES`.

Nel primo passaggio, viene creato il file `.EDGES` dove ogni riga rappresenta un arco nel grafo, indicando i nodi connessi. Nel secondo passaggio, vengono generati casualmente dei cluster e assegnati i nodi ad essi. I nodi vengono distribuiti casualmente tra cinque cluster, e queste informazioni vengono salvate nel file `.CIRCLES`, dove ogni riga rappresenta un cluster e i nodi ad esso assegnati. Questi due file sono utilizzati successivamente per la costruzione del grafo all'interno del programma.

```
import networkx as nx
import random

n = 100
p = 0.05

G = nx.erdos_renyi_graph(n, p)

# Salva gli archi del grafo nel file .EDGES
with open('networks/generated_networks/graph.EDGES', 'w') as edges_file:
    for edge in G.edges():
        edges_file.write(f"{edge[0]} {edge[1]}\n")

# Assegna casualmente i nodi ai cerchi e salva queste informazioni nel file .CIRCLES
circles = {}

for node in G.nodes():
    circle_id = f"circle_{random.randint(1, 5)}"  # Assegna casualmente il nodo a uno dei 5 cerchi
    if circle_id not in circles:
        circles[circle_id] = []
    circles[circle_id].append(str(node))

with open('networks/generated_networks/graph.CIRCLES', 'w') as circles_file:
    for circle_id, members in circles.items():
        circles_file.write(f"{circle_id} {' '.join(members)}\n")
```

2.9 Script Handler

Eseguire lo script avvia un handler ("main") che, per prima cosa, individua un seed set massimale utilizzando l'algoritmo di diffusione specificato dall'utente, che - appunto - si occupa di individuare il seed set massimale tale che la funzione di costo associata sia minore di un certo budget. Il seed set individuato viene quindi stampato a schermo. In seguito, viene creato un oggetto `CascadeProcess` per simulare il processo di diffusione nel

grafo utilizzando il seed set precedentemente identificato. L'output di questa simulazione sono i nodi influenzati dal processo di diffusione, inclusi quelli nel seed set, che vengono stampati a schermo. Infine, se l'opzione di salvataggio dei risultati è attiva, i risultati vengono salvati in una cartella specificata.

```
if __name__ == "__main__":
    spreading_process: SpreadingProcess = SpreadingProcess(options =
    ↪  setup_options())

    """
    Individua il seed set massimale tale che la funzione di costo c(S)
    sia minore o uguale al threshold k.
    """
    seed_set: list = \
         spreading_process   \
        .spreading_algorithm \
        .get_seed_set()

    print(f"Seed Set: {seed_set}")

    """
    Simula il processo di diffusione a partire dal grafo G
    e il seed set S individuato in precedenza,
    restituendo i nodi influenzati al termine
    del processo (inclusivi dei nodi appartenenti al seed set).
    """
    cascade_process: CascadeProcess = CascadeProcess(
        graph = spreading_process.G,
        seed_set = seed_set,
        verbose = spreading_process.options.verbose
    )

    influenced_nodes: list = \
         cascade_process \
        .get_influenced_nodes()

    print(f"Influenced Nodes: {influenced_nodes}")

    """
    Persiste i risultati in una cartella.
    """
    if spreading_process.options.save:
        save_results(
            file_path = f"{spreading_process.results_path}",
            results = f"Seed Set: {len(seed_set)} \
```

```
            Influenced Nodes: {len(influenced_nodes)}",
            graph = spreading_process.G,
            seed_set = seed_set,
            influenced_nodes = influenced_nodes
        )
```

2.9.1 CLI Parser

Questa funzione è progettata per configurare e gestire le opzioni della linea di comando dello script, utilizzando la libreria `argparse` per facilitare l'interpretazione e l'elaborazione degli argomenti forniti dall'utente. Inizialmente, viene creato un parser di argomenti con `argparse.ArgumentParser()`, che consente di definire una serie di opzioni configurabili dall'utente.

```
# Consultare il README per le possibili configurazioni dello script
def setup_options() -> None:
        parser: argparse.ArgumentParser = argparse.ArgumentParser()

        parser.add_argument('-g', '--print_graph', action =
        ↪  'store_true')
        parser.add_argument('-v', '--verbose', action = 'store_true')
        parser.add_argument('-s', '--save', action = 'store_true')

        parser.add_argument('-k', '--threshold', default =
        ↪  config.DEFAULT_THRESHOLD)
        parser.add_argument('-e', '--edges', default =
        ↪  config.DEFAULT_EDGES)
        parser.add_argument('-c', '--circles', default =
        ↪  config.DEFAULT_CIRCLES)

        parser.add_argument('-cf', '--cost_function', default =
        ↪  config.DEFAULT_COST_FUNCTION, choices = ['1', '2', '3'])
        parser.add_argument('-sf', '--submodular_function', default =
        ↪  config.DEFAULT_SUBMODULAR_FUNCTION, choices = ['1', '2',
        ↪  '3'])
        parser.add_argument('-a', '--algorithm', default =
        ↪  config.DEFAULT_ALGORITHM, choices = ['1', '2', '3'])

        args = parser.parse_args()

        print(f"Caricati i nodi da {args.circles}")
        print(f"Caricati gli archi da {args.edges}")

```

```
22          return args
```

2.9.2 Classe SpreadingProcess

La classe `SpreadingProcess` è progettata per gestire il processo di diffusione nel grafo, configurato secondo le opzioni specificate dall'utente tramite la CLI. Durante l'inizializzazione, vengono impostati i parametri fondamentali come il grafo, la soglia di diffusione (budget k), la funzione di costo e l'algoritmo di diffusione. Il percorso per salvare i risultati è generato dinamicamente, includendo i parametri selezionati per garantire tracciabilità e riproducibilità.

La funzione `setup_graph` legge i file degli archi e delle cerchie specificati dall'utente per costruire il grafo. Se l'opzione `-g` o `--print_graph` è attivata, il grafo viene visualizzato utilizzando Matplotlib. La funzione `setup_cost_function` seleziona una funzione di costo basata sull'opzione `-cf` o `--cost_function`, utilizzando una factory per ottenere la funzione appropriata. Infine, il metodo `setup_spreading_algorithm` configura l'algoritmo di diffusione secondo l'opzione `-a` o `--algorithm`, tenendo conto della funzione submodulare specificata. Questi metodi garantiscono che il processo di diffusione sia personalizzato e adattabile a diverse esigenze di sperimentazione.

```
1  class SpreadingProcess():
2      def __init__(self, options):
3          self.options = options
4
5          self.G: Graph = self.setup_graph()
6          self.k: int = int(self.options.threshold)
7          self.cost_function: callable = self.setup_cost_function()
8          self.spreading_algorithm: SpreadingAlgorithm =
           ↪  self.setup_spreading_algorithm()
9
10         self.results_path =
           ↪  f"{config.DEFAULT_RESULTS_PATH}/{time.time()}_k_{self.k}\
11         _a_{self.options.algorithm}_cf_{self.options.cost_function}"
12
13
14     # Per selezionare liste di nodi ed archi differenti, usare le
       ↪  opzioni -c (--circles) ed -e (--edges)
15     def setup_graph(self) -> Graph:
16         graph: Graph = nl.read_network(
17             edges_file_path = self.options.edges,
18             circles_file_path = self.options.circles
19     )
```

```
20
21          # Per stampare il grafo, usare l'opzione -g oppure --print_graph
22          if self.options.print_graph:
23              nx.draw(G = graph, pos = nx.spring_layout(graph),
                ↪  with_labels = True)
24              plt.show(block = False)
25
26          return graph
27
28
29      # Per selezionare una funzione di costo, usare l'opzione -cf oppure
        ↪  --cost-function, es. -cf 1 (valori ammessi: 1, 2, 3)
30      def setup_cost_function(self) -> callable:
31          selected_algorithm_index: int = int(self.options.cost_function)
            ↪  # Rappresenta la funzione di costo scelta (1, 2 o 3)
32
33          cost_function_factory: cf.CostFunctionFactory =
            ↪  cf.CostFunctionFactory(
34              selected_algorithm_index = selected_algorithm_index,
35              graph = self.G,
36              range_min = config.DEFAULT_RANGE_MIN,
37              range_max = config.DEFAULT_RANGE_MAX,
38              d_max = max(dict(self.G.degree()).values()),
39              verbose = self.options.verbose
40          )
41
42          return cost_function_factory.get_function()
43
44
45      # Per selezionare un algoritmo, usare l'opzione -a oppure
        ↪  --algorithm, es. -a 1 (valori ammessi: 1, 2, 3)
46      def setup_spreading_algorithm(self) -> SpreadingAlgorithm:
47          selected_algorithm_index: int = int(self.options.algorithm) #
            ↪  Rappresenta l'algoritmo scelto (1, 2 o 3)
48
49          """
50          Questa variabile rappresenta la funzione submodulare selezionata
51          tramite l'opzione -sf oppure --submodular--function
52          (valori ammessi: 1, 2, 3). La funzione entra in gioco solo
53          nel caso dell'algoritmo Cost-Seeds-Greedy,
54          ma viene ugualmente pre-impostato ad un valore di default.
55          """
56          selected_submodular_function_index: int =
            ↪  int(self.options.submodular_function)
```

```
57
58          spreading_algorithm: SpreadingAlgorithm = SpreadingAlgorithm(
59              selected_algorithm_index = selected_algorithm_index,
60              selected_submodular_function_index =
                ↪  selected_submodular_function_index,
61              graph = self.G,
62              threshold = self.k,
63              cost_function = self.cost_function,
64              verbose = self.options.verbose
65          )
66
67          return spreading_algorithm
```

2.9.3 Configurazione programmatica

E' possibile configurare i parametri di default dello script modificando il file `default_config.py`. Le opzioni di default, indicate nel file, sono sovrascribili mediante CLI, come indicato nella sezione successiva.

```
1  from enum import Enum
2
3  # Configurazioni di default, sovrascrivibili tramite CLI
4
5  DEFAULT_THRESHOLD = 6
6
7  DEFAULT_EDGES = 'networks/generated_networks/graph.EDGES'
8  DEFAULT_CIRCLES = 'networks/generated_networks/graph.CIRCLES'
9
10 DEFAULT_RANGE_MIN = 1
11 DEFAULT_RANGE_MAX = 10
12
13 DEFAULT_COST_FUNCTION = 1
14 DEFAULT_SUBMODULAR_FUNCTION = 1
15 DEFAULT_ALGORITHM = 1
16
17 DEFAULT_RESULTS_PATH = "results"
18
19 class Algorithms(Enum):
20     COST_SEEDS_GREEDY = 1
21     WTSS = 2
22     MY_SEEDS = 3
```

2.9.4 Salvare i risultati

Questa semplice utility permette di persistere i risultati e salvarli in una cartella. In particolare, genera dei grafici appena generato il grafo, dopo aver individuato il seed set, e dopo aver effettuato il processo di influenza, colorando in modo diverso i rispettivi nodi del grafo.

```
def save_results(file_path, results, graph, seed_set, influenced_nodes):
    if not os.path.exists(file_path):
        os.makedirs(file_path)

    with open(f"{file_path}/data.txt", "a") as file:
        file.write(f"{results}\n")

    """
    Grafico della rete senza particolari colorazioni
    """
    color_map = []

    for node in graph.nodes():
        color_map.append('gray')

    nx.draw(G = graph, with_labels = False, pos =
    ↪  nx.spring_layout(graph, scale = 3), node_color = color_map,
    ↪  node_size = 10)
    plt.savefig(f"{file_path}/pre-influencing.png", format = "PNG")
    plt.clf()

    """
    Grafico che evidenzia i nodi del seed set (rossi)
    e i nodi rimanenti (grigio)
    """
    color_map = []

    for node in graph.nodes():
        if node in seed_set:
            color_map.append('red')
        else:
            color_map.append('gray')

    nx.draw(G = graph, with_labels = False, pos =
    ↪  nx.spring_layout(graph, scale = 3), node_color = color_map,
    ↪  node_size = 10)
    plt.savefig(f"{file_path}/influencing.png", format = "PNG")
    plt.clf()
```

```
35
36      """
37      Grafico che evidenzia i nodi del seed set (rossi),
38      i nodi influenzati dal seed set (blu)
39      e i nodi rimanenti (grigio)
40      """
41      color_map = []
42
43      for node in graph.nodes():
44          if node in seed_set:
45              color_map.append('red')
46          elif int(node) in influenced_nodes:
47              color_map.append('blue')
48          else:
49              color_map.append('gray')
50
51      nx.draw(G = graph, with_labels = False, pos =
        ↪  nx.spring_layout(graph, scale = 3), node_color = color_map,
        ↪  node_size = 10)
52      plt.savefig(f"{file_path}/influencing_and_influenced.png", format =
        ↪  "PNG")
53      plt.clf()
54
55      print(f"Risultati salvati in {file_path}")
```

3 Installazione ed utilizzo

Il primo comando eseguito è `conda create -n rs python=3.7.0`. Questo comando utilizza Conda, un gestore di pacchetti e ambienti per Python, per creare un nuovo ambiente virtuale denominato rs. Specificando `python=3.7.0`, si impone l'installazione della versione 3.7.0 di Python all'interno di questo ambiente. La scelta di una versione specifica di Python è cruciale per assicurare la compatibilità con i pacchetti e le librerie che verranno successivamente installati, evitando potenziali conflitti dovuti a differenze di versione.

Successivamente, l'ambiente appena creato viene attivato con il comando `conda activate rs`. L'attivazione dell'ambiente rs fa sì che tutte le operazioni successive nel terminale avvengano all'interno di questo ambiente isolato. Ciò permette di gestire le dipendenze specifiche del progetto senza interferire con altre versioni di pacchetti o librerie installate globalmente o in altri ambienti.

Infine, il comando `pip install -r requirements.txt` viene eseguito per installare tutte le dipendenze elencate nel file `requirements.txt`. Il file requirements.txt contiene una lista di pacchetti necessari per il progetto, ognuno specificato con la sua versione.

```
conda create -n rs python=3.7.0
conda activate rs
pip install -r requirements.txt
```

```
python main.py [-h] [-g] [-v] [-s] [-k THRESHOLD] [-e EDGES] [-c
 ↪  CIRCLES] [-cf {1,2,3}] [-sf {1,2,3}] [-a {1,2,3}]
```

La CLI offre una serie di opzioni che permettono di eseguire il programma principale `main.py` con diversi parametri e modalità operative. Di seguito vengono illustrati i vari comandi utilizzabili, insieme alle loro rispettive funzioni.

L'esecuzione standard del programma, che avvia il processo di diffusione con i parametri di default, si ottiene semplicemente digitando `python main.py`. Questa modalità è utilizzata per eseguire l'algoritmo principale senza specificare ulteriori opzioni, sfruttando i valori predefiniti per tutti i parametri.

Per abilitare la modalità verbosa, che consente di ottenere informazioni dettagliate durante l'esecuzione del programma, si utilizza il comando `python main.py -v`. Questa opzione è utile per il debug e per monitorare in dettaglio il comportamento dell'algoritmo, fornendo un output più ricco di informazioni rispetto all'esecuzione standard.

L'opzione `-g` attiva la stampa di debug del grafo, mentre `-s` permette di salvare i risultati dell'esecuzione. Queste due opzioni possono essere combinate, come mostrato nel comando `python main.py -g -s`, che consente di visualizzare le informazioni dettagliate sul grafo utilizzato e di salvare i risultati prodotti dall'algoritmo.

Per utilizzare un grafo personalizzato, è possibile specificare i file contenenti le informazioni sulle connessioni e i nodi del grafo mediante le opzioni `-e` e `-c`.

La CLI permette anche di selezionare diversi algoritmi per l'individuazione del set di semi (seed set), utilizzando l'opzione `-a` seguita da un numero identificativo dell'algoritmo. Ad esempio, `python main.py -a=1` utilizza l'algoritmo Cost-Seeds-Greedy, `python main.py -a=2` utilizza l'algoritmo WTSS, mentre `python main.py -a=3` utilizza l'algoritmo My-Seeds. Questa flessibilità consente di confrontare le performance di diversi algoritmi sotto le stesse condizioni.

Inoltre, è possibile selezionare specifiche funzioni di costo e funzioni submodulari mediante le opzioni `-cf` e `-sf`. Ad esempio, il comando `python main.py -cf=1` seleziona i costi randomizzati come funzione di costo, mentre `python main.py -sf=2` seleziona la seconda funzione submodulare. È possibile combinare queste opzioni come illustrato nel comando `python main.py -cf=1 -sf=3`, che utilizza costi randomizzati e la terza funzione submodulare.

```
### Esegue il processo di diffusione con i parametri di default
python main.py

### Abilita la modalità verbosa
python main.py -v

### Abilita la stampa di debug del grafo e salva i risultati
python main.py -g -s

### Seleziona ed utilizza un grafo personalizzato
python main.py -e=networks/sample_networks/0.edges
 ↪  -c=networks/sample_networks/0.circles

### Seleziona uno specifico algoritmo di individuazione del seed set
python main.py -a=1 # Cost-Seeds-Greedy
python main.py -a=2 # WTSS
python main.py -a=3 # My-Seeds

### Seleziona specifiche funzioni di costo e funzioni submodulari (es.
 ↪  1, 2, 3)
python main.py -cf=1 # Random Costs
python main.py -sf=2 # Second Submodular Function (ref.
 ↪  Costs-Seeds-Greedy)
python main.py -cf=1 -sf=3 # Random Costs & Second Submodular Function
```

3.1 Dipendenze e librerie

Le librerie Python utilizzate per il progetto sono state annotate con le rispettive versioni in un pratico file *requirements.txt* nella directory del progetto. E' possibile installarle tramite l'utility PIP. Si noti che queste dipendenze sono fondamentali per eseguire il progetto.

```
cycler==0.11.0
fonttools==4.38.0
kiwisolver==1.4.5
matplotlib==3.5.3
networkx==2.6.3
numpy==1.21.6
packaging==24.0
Pillow==9.5.0
pyparsing==3.1.2
python-dateutil==2.9.0.post0
scipy==1.7.3
six==1.16.0
typing_extensions==4.7.1
```

4 Risultati e conclusioni

Ogni algoritmo, per stesso valore di budget k e stessa funzione di costo $c : V \to \mathbb{N}$ (e per stessa funzione submodulare $f : 2^V \to \mathbb{R}^+$, nel caso dell'algoritmo *Cost-Seeds-Greedy*) è stato eseguito 10 volte. La sperimentazione è consistita nello studiare, al variare del budget k, il numero medio di nodi influenzati (la taglia media dell'insieme *Inf*) dal Seed-Set generato dall'algoritmo specificato, sulla funzione di costo specificata (e per la funzione submodulare specificata, nel caso dell'algoritmo *Cost-Seeds-Greedy*).

Gli algoritmi utilizzati sono:

- CSG (Cost Seeds Greedy)
- WTSS (Weighted Target Set Selection)
- My-Seeds (l'algoritmo ideato)

Le funzioni di costo $c : V \to \mathbb{N}$ sono:

- $c(v) \sim U(1, 10)$
- $c(v) = \lceil \frac{d(v)}{2} \rceil$
- $c(v) = \lceil \frac{d(v)^2}{d_{\max}} \rceil$

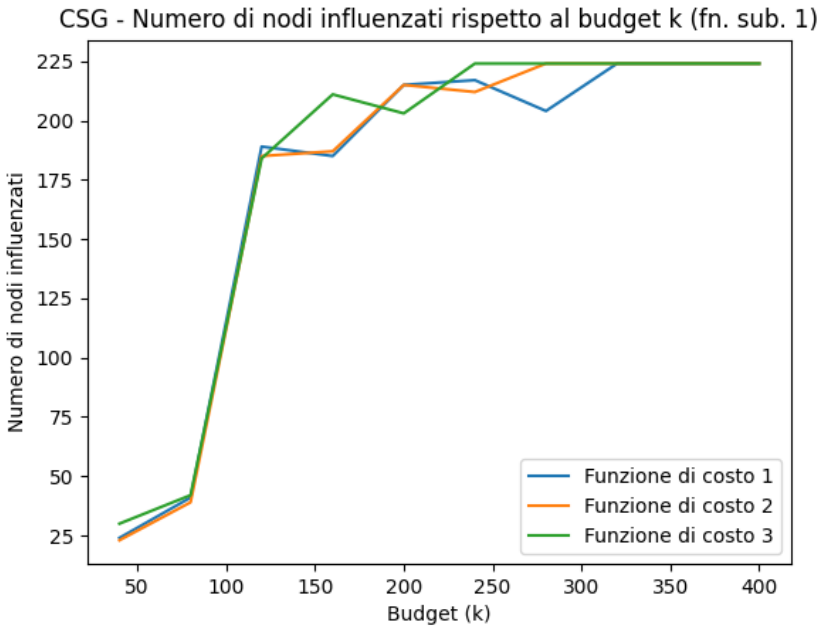
Nel caso specifico dell'algoritmo Costs-Seeds-Greedy, le funzioni submodulari sono:

- $f_1(S) = \sum_{v \in V} \min\{|N(v) \cap S|, \lceil \frac{d(v)}{2} \rceil\}$
- $f_2(S) = \sum_{v \in V} \sum_{i=1}^{|N(v) \cap S|} \max\left\{\lceil \frac{d(v)}{2} \rceil - i + 1, 0\right\}$
- $f_3(S) = \sum_{v \in V} \sum_{i=1}^{|N(v) \cap S|} \max\left\{\frac{\lceil \frac{d(v)}{2} \rceil - i + 1]}{d(v) - i + 1}, 0\right\}$

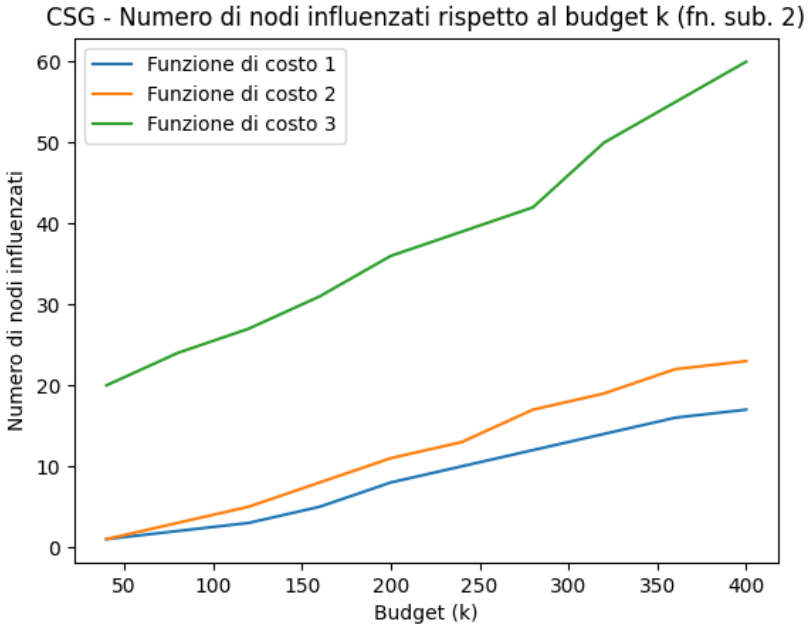
4.1 Risultati per Cost Seeds Greedy

Nei seguenti grafici è possibile osservare, rispettivamente per la prima funzione submodulare, la seconda e la terza, la size media di *Inf* al variare del budget k, per le tre diverse funzioni di costo (i tre colori diversi). L'algoritmo utilizzato è Cost-Seeds-Greedy.

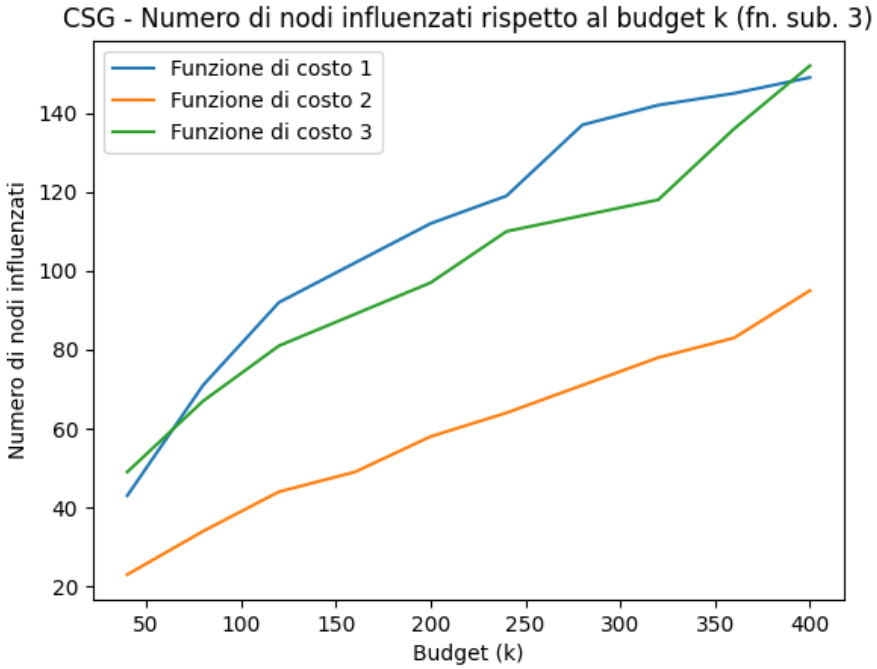
È possibile osservare, per ogni funzione submodulare, per ogni funzione di costo utilizzata, un andamento pressochè crescente per ogni funzione. La size media dell'insieme dei nodi influenzati aumenta con l'aumentare del budget.



Size media di Inf al variare di k (SF1)



Size media di Inf al variare di k (SF2)



Size media di Inf al variare di k (SF3)

I risultati migliori si ottengono con l'utilizzo della **prima funzione submodulare**, dove a prescindere alla funzione di costo, a partire da un budget di circa 300, il seed-set generato è in grado di raggiungere una complete cascade, influenzando tutti i nodi della rete.

I risultati peggiori sono ottenuti in corrispondenza dell'utilizzo della **seconda funzione submodulare** dove anche il budget massimo utilizzato per la sperimentazione (k = **400**) genera seed-sets in grado di influenzare in media circa **60** nodi (compresi i nodi appartenenti ai seed-set stessi), appena poco più del 25% dei nodi totali della rete (**224** nodi).

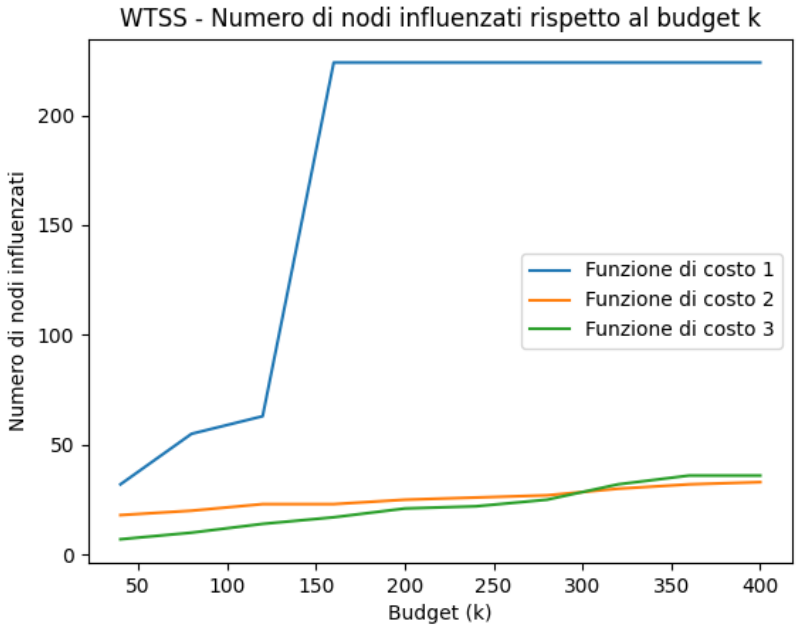
I grafici relativi alle funzioni submodulari 2 e 3 condividono una **natura lineare**, aventi offset diversi a seconda della funzione di costo specificata e quasi tutte lo stesso coefficiente angolare.

Differente la **natura quasi logaritmica** presentata dalle funzioni di costo nella sperimentazione dove è fissata la funzione submodulare 1.

4.2 Risultati per Weighted Target Set Selection

Nel seguente grafico è possibile osservare la size media di *Inf* al variare del budget k, per le tre diverse funzioni di costo (i tre colori diversi). L'algoritmo utilizzato è Weighted Target Set Selection.

È possibile osservare come, per lo stesso algoritmo WTSS, funzioni di costo diverse



Size media di Inf al variare di k

mostrano funzioni con comportamenti diversi. Tutte le funzioni hanno una natura crescente ma non lo stesso andamento; in particolare, per quanto riguarda la funzione di costo 1, in media è possibile raggiungere una complete cascade con un budget limitato, già a partire da k = **160** circa.

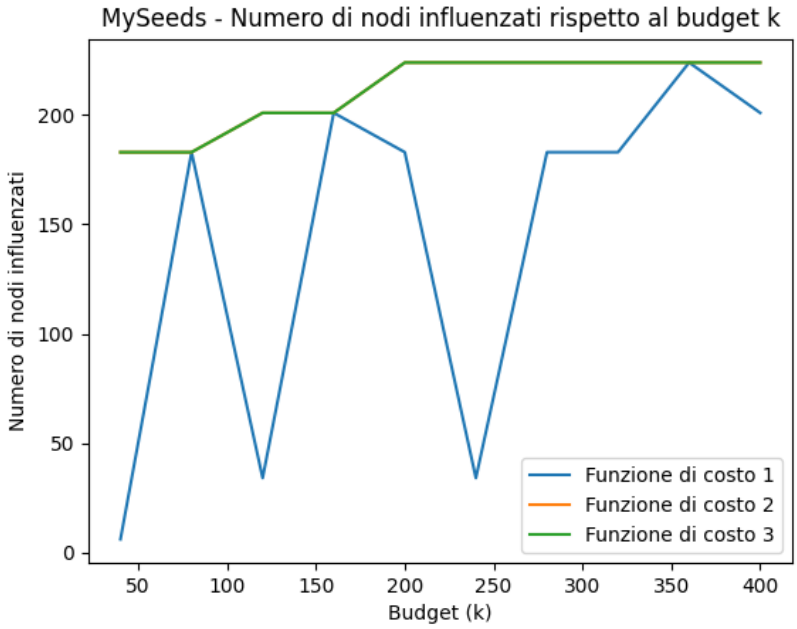
Per la funzione 1 è possibile osservare un rapido incremento della size media di Inf a partire da budget appena più grandi di k = **140**, fenomeno che non è invece possibile osservare quando invece è utilizzata la seconda o la terza funzione di costo, che hanno un andamento lineare e non riescono ad influenzare più di 30 nodi anche avendo a disposizione il massimo budget (k = **400**).

4.3 Risultati per MySeeds

Nel seguente grafico è possibile osservare la size media di *Inf* al variare del budget *k*, per le tre diverse funzioni di costo (i tre colori diversi). L'algoritmo utilizzato è l'algoritmo ideato per il progetto.

Ciò che è possibile osservare è che la funzione associata all'uso della funzione di costo 2 non è visibile nel grafico; in realtà essa ha lo stesso esatto andamento della funzione associata all'uso della funzione di costo 3, quindi le funzioni si accavallano graficamente.

L'andamento della funzione associata alla funzione di costo 2 e 3 è crescente all'aumentare del budget k; budget k=240 sono sufficienti per generare un seed-set che in media porta alla complete cascade nel processo di influenza.



Size media di Inf al variare di k

L'andamento della funzione associata alla funzione di costo 1 è altalenante, con momenti di crescita e momenti di decrescita, molto ripidi.

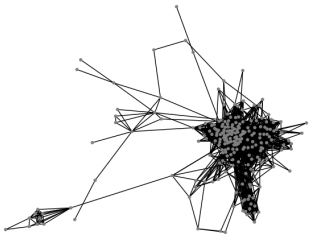
4.4 Processo di influenza: esempi di grafi

Di seguito vengono allegati i grafici dei grafi generati tramite il processo di diffusione. In particolare, ogni riga rappresenta una tripletta di grafici in cui il primo grafico è una semplice rappresentaizone del grafo in cui i nodi sono colorati di grigio; il secondo grafico è una rappresentazione in cui i nodi afferenti al seed set (individuati mediante uno dei tre algoritmi considerati) è colorato in rosso, e i restanti nodi sono colorati in grigio; il terzo grafico è una rappresentazione in cui i nodi afferenti al seed set sono colorati in rosso, e i nodi influenzati ("Inf") dal seed set sono colorati in blu, mentre i restanti nodi sono colorati in grigio.

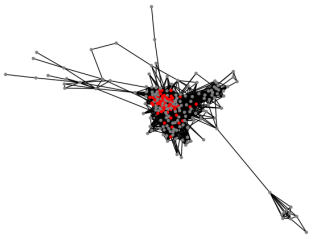
Queste rappresentazioni grafiche, di cui di seguito se ne rilascia solo una parte, mentre è possibile trovare ulteriori grafici sulla repository del progetto nella directory "graphs", permettono di capire quale configurazione riesce a raggiungere e quindi influenzare il maggior numero di nodi.

In particolare, si presentano grafici relativi a sperimentazioni legati alla rete 348, fissando la funzione di costo alla prima, e facendo variare il budget K (160, 320), e usando tutti gli algoritmi a disposizioni, per un totale di oltre 18 grafici. E' possibile produrre ulteriori grafici usando la CLI del programma, come descritto nelle sezioni precedenti.

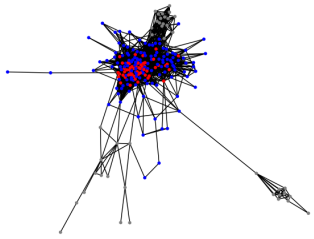
Ad esempio, dai grafici è possibile osservare che l'algoritmo MySeeds performi nettamente meglio in un contesto ad alto budget, mentre gli algoritmi WTSS e CSG riescono in ogni caso a penetrare notevolmente la rete. Si nota che i grafici per K = 320 sono disponibili su GitHub.



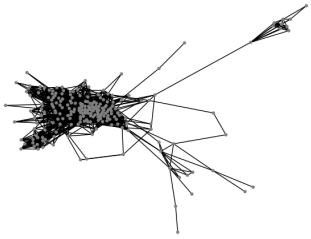
(a) Rete 348, K=160, CF=1, Alg. CSG - Grafo di base



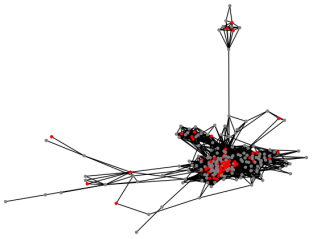
(b) Rete 348, K=160, CF=1, Alg. CSG - Seed Set



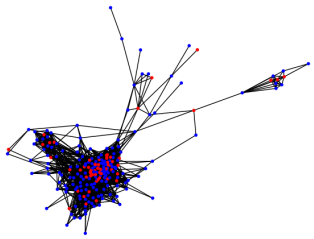
(c) Rete 348, K=160, CF=1, Alg. CSG - Seed Set & Influenced



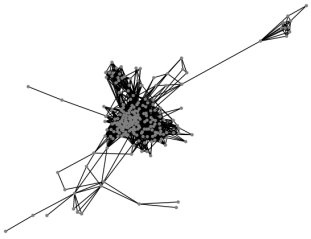
(d) Rete 348, K=160, CF=1, Alg. WTSS - Grafo di base



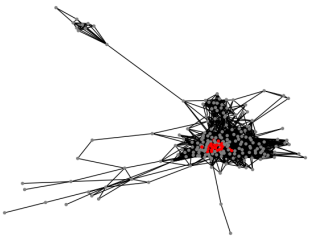
(e) Rete 348, K=160, CF=1, Alg. WTSS - Seed Set



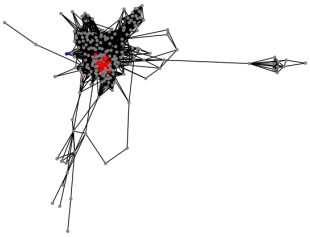
(f) Rete 348, K=160, CF=1, Alg. WTSS - Seed Set & Influenced



(g) Rete 348, K=160, CF=1, Alg. MySeeds - Grafo di base



(h) Rete 348, K=160, CF=1, Alg. MySeeds - Seed Set



(i) Rete 348, K=160, CF=1, Alg. MySeeds - Seed Set & Influenced

4.5 Conclusioni

Dall'analisi dei grafici si può appurare che, oltre al valore k del budget a disposizione per la generazione del seed-set, anche l'algoritmo utilizzato incide fortemente sulla dimensione media dell'insieme dei nodi influenzati. Il Cost-Seeds-Greedy, a parità di budget, è

l'algoritmo che genera risultati migliori e più stabili rispetto agli altri algoritmi utilizzati ed implementati.

5 Bibliografia e riferimenti

1. SNAP Datasets: Stanford Large Network Dataset Collection
2. Matplotlib
3. Networkx
4. Ego-Facebook Dataset
5. Facebook Graph API