Progetto realizzato da: Marco Delle Cave, Andrea Tranquillo, Santolo Mutone

Dynamics in signed threshold networks

1. Il Problema

Il problema del Dynamics in signed threshold networks consiste nel trovare all'interno di una signed networks, ovvero una rete utilizzata per modellare e analizzare reti sociali complesse in cui le relazioni possono essere sia positive che negative (ad esempio nelle reti di collaborazione scientifica, nelle reti di conflitto o nelle reti di relazioni interpersonali), un insieme di nodi k che possa massimizzare il numero di vertici attivi all'interno della rete.

Formalmente una signed network può essere modellata attraverso un grafo non direzionato G = (V, E^+ U E^-), dove per ogni coppia di vertici u e v del grafo:

- (u,v) ∈ E^+ significa che c'è una interazione **positiva** tra u e v
- (u,v) ∈ E^- significa che c'è una interazione **negativa** tra u e v

Per ogni vertice v della rete è stato assegnato un valore **threshold** costante, dove quest'ultimo è definito come il numero minimo di adiacenti di v necessari per influenzare tale nodo.

Il problema dell'influenza è un tipico problema di "influenza sociale". È un processo in cui un individuo modifica le proprie opinioni o i propri comportamenti tramite l'interazione con altre persone. Quando gli individui diventano consapevoli di nuove idee o prodotti, essi tendono a diffonderlo ai loro amici. Inoltre, le persone, in genere, tendono sempre a conformarsi, per fare in modo da non sentirsi esclusi dai vari gruppi di persone che si vengono a formare.

L'idea di questa attività progettuale è quella di trovare il numero di persone minimo che possano influenzare tutte le altre. Rapportando questo problema con i grafi e le reti sociali, bisogna trovare il numero minimo di nodi che riescano ad influenzare l'intera rete. I nodi che rispettano tali condizioni vengono inseriti in un insieme che viene chiamato **Seed Set**.

Più formalmente, quindi, il problema della diffusione dell'influenza può essere descritto nel seguente modo:

Data una rete G = (V,E), una funzione di threshold t:V -> {1,2, …} ed un seed set S ⊆ V, un processo dinamico di diffusione dell'influenza è definita come la sequenza di un sottoinsieme di nodi:

Influenced[S, 0] , Influenced[S, 1] , …, Influenced[S, r] , …, $\subseteq V$

dove:

- $\text{Influenced}[S, 0] = S$
- $\text{Influenced}[S, r] = \text{Influenced}[S, r\text{-}1] \cup \{v : |N(v) \cap \text{Influenced}[S, r\text{-}1]| \geq t(v)\}$ con $N(v) =$ insieme di nodi adiacenti di v

Il lavoro da noi svolto ha dunque l'obiettivo di confrontare, a partire da una signed networks, diversi algoritmi per la determinazione del seed set, e valutare quali tra questi offre performance migliori per quanto riguarda la massimizzazione dell'influenza sull'intera rete.

Il tutto può essere riassunto nei seguenti step:

- Dato un grafo **G=(V,E)** ed una threshold function **t**: V -> {1,2,…},
- Dato un intero k
- Data una distribuzione di probabilità associata agli archi di G **p**:E -> [0,1] I. Si costruisce $G_p = (V, E^+ \cup E^-)$

II. Si applica **Algorithm x** (2 dei 4 algoritmi che sono stati forniti + l'algoritmo da noi ideato) per determinare il seed set S_p con $|S_p| = k$

III. Si fa girare il processo di attivazione e si determina l'insieme dei nodi attivati $\text{Inf}[S_p]$

- Si ripetono per 10 volte gli step 1. 2. e 3. E viene poi calcolata la media $\boldsymbol{\sigma}$ della size dell'insieme dei nodi attivati

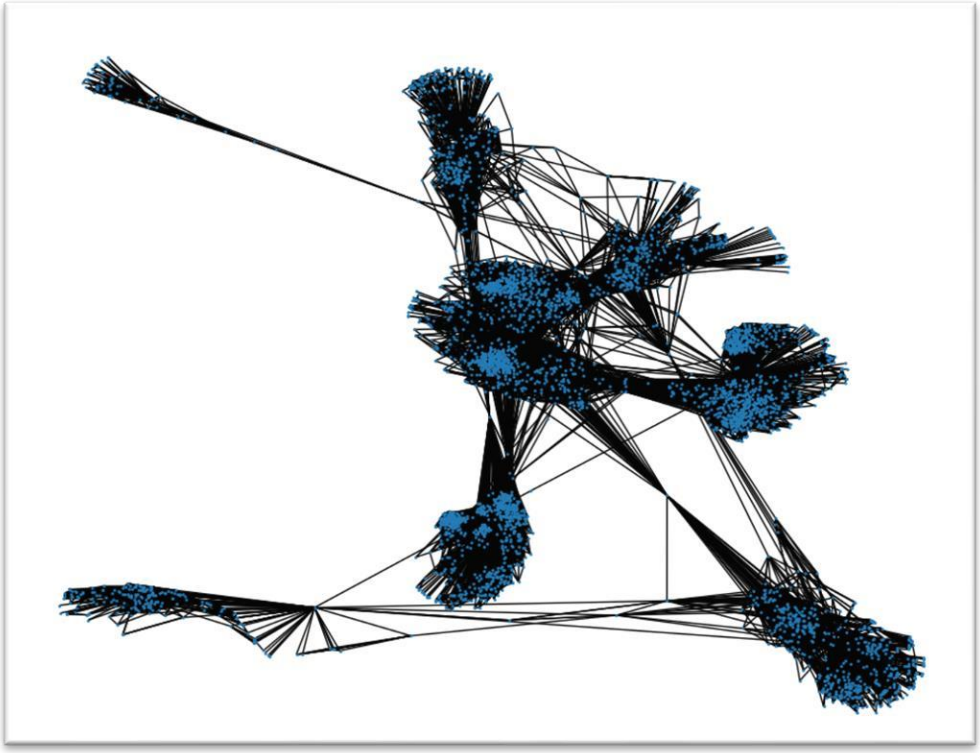
2. La Rete Scelta

La rete scelta è "**ego-Facebook**", una delle reti messe a disposizione da **SNAP** (Stanford Network Analysis Project) all'interno del loro repository pubblico.

Il dataset scelto consiste nelle "cerchie" (o friend-lists) di Facebook. I dati sono stati raccolti tramite i partecipanti ad un sondaggio utilizzando l'app di Facebook. Il dataset include le caratteristiche dei nodi (profili), le cerchie e gli *ego networks*, ovvero reti costituite di un nodo focale ("ego"), dei nodi ai quali l'ego è direttamente connesso (detti "alters") e dei collegamenti, se presenti, tra gli alters.

I dati sono stati anonimizzati rimpiazzando gli id interni di Facebook con un nuovo valore. Inoltre, dal momento che sono stati forniti i vettori contenenti le varie features di questo dataset, l'interpretazione di queste caratteristiche è stata anonimizzata. Per esempio, dove nel dataset

originale era presente la caratteristica "*political=Democratic Party",* il nuovo valore per questa feature conterrebbe semplicemente *"political=anonymized feature 1".* Con i dati anonimizzati, è possibile determinare se due utenti hanno le stesse affiliazioni politiche, ma non è possibile sapere le effettive affiliazioni dei singoli individui.



Caratteristiche della rete	Nodi	Archi	Coefficiente di clustering medio	Grado medio
Valori	**4039**	**88234**	**0.6055**	**43.69**

La scelta è ricaduta su questo dataset principalmente per tre motivi in particolare:

- Si tratta di una rete densamente connessa in generale, e questo si evince dal coefficiente di clustering medio e dal valore di grado medio dei nodi
- E' una rete divisa in comunità abbastanza grandi ed allo stesso tempo fortemente connesse al loro interno, e questo si evince dalla rappresentazione visuale del grafo
- E' una rete che si adatta bene al problema da noi analizzato, ovvero le signed networks, poiché si tratta di una rete sociale a tutti gli effetti dove possono intercorrere tra i nodi sia relazioni positive sia relazioni negative

3. Dettagli implementativi

Per lavorare con il dataset disponibile su SNAP, abbiamo utilizzato il linguaggio **Python** e il modulo **Snap.py**, il quale ci ha permesso non solo di caricare il grafo, ma anche di accedere alla varie informazioni utili ai fini dell'algoritmo, come il degree dei nodi.

3.1 Caricamento del grafo

Abbiamo utilizzato la funzione LoadEdgeList(…) di SNAP, che ci ha consentito di importare il grafo da un file di testo. Tra i parametri desiderati troviamo:

- **Tipo del grafo**: ovvero il tipo di grafo che si vuole ottenere. In questo caso è stato scelto un TUNGraph, ovvero un grafo non direzionato.
- **Nome del file**: il nome del file da cui importare il grafo
- **Colonna dei nodi sorgente**: viene specificata la colonna del file in cui sono presenti gli id dei nodi sorgente
- **Colonna dei nodi destinazione**: viene specificata la colonna del file in cui sono presenti gli id dei nodi destinazione

```
G2 = snap.LoadEdgeList(snap.TUNGraph, "facebook_combined.txt", 0, 1)
```

3.2 Edge Labeling

Dato che il nostro obiettivo è quello di lavorare con una signed network, il primo passo è stato quello di etichettare ogni arco del grafo con un segno positivo o negativo. L'etichettatura degli archi del grafo G=(V,E) segue una distribuzione di probabilità associata ad ogni arco **p**:E -> [0,1]:

- Probabilità p(u,v) per (u,v) ∈ E = 1/max{d(u),d(v)}, dove d(v) indica il grado del nodo v

```python
def edge_labeling(G2):

    random.seed()
    signed_edges=[]

    for EI in G2.Edges():
        if EI.GetSrcNId() == EI.GetDstNId():
            continue
        value = random.random()
        if value <= getMaxDegree(G2,EI.GetSrcNId(),EI.GetDstNId()):
            signed_edges.append((EI.GetSrcNId(),EI.GetDstNId(),'-'))
        else:
            signed_edges.append((EI.GetSrcNId(),EI.GetDstNId(),'+'))

    return signed_edges
```

Figura 3-1 Algoritmo con probabilità proporzionale

- Probabilità p(u,v) per (u,v) ∈ E costante

```python
def edge_labeling(G2):

    random.seed()
    signed_edges=[]

    for EI in G2.Edges():
        if EI.GetSrcNId() == EI.GetDstNId():
            continue
        value = random.random()
        if value <= 0.01:
            signed_edges.append((EI.GetSrcNId(),EI.GetDstNId(),'-'))
        else:
            signed_edges.append((EI.GetSrcNId(),EI.GetDstNId(),'+'))

    return signed_edges
```

Figura 3-2 Algoritmo con probabilità costante

3.3 Implementazione degli algoritmi

Per quanto riguarda l'identificazione dei nodi che andranno a comporre il Seed Set iniziale, sono stati implementati 2 dei 3 algoritmi proposti, ed in più l'algoritmo da noi ideato.

3.3.1 Seeds-Greedy -Difference max (G,k) – "3rd Algorithm"

Il primo algoritmo che è stato implementato è l'algoritmo "Seeds-Greedy-Difference max", che, ad ogni step, aggiunge all'interno del Seed Set il nodo con $\mathbf{d^+(v) \geq d^-(v)}$, che massimizza il rapporto $\mathbf{(d^+(v) - d^-(v)) / t(v)}$. Dopodiché aggiorna il grado positivo e negativo di tutti i vicini del nodo aggiunto all'interno del Seed Set.

L'operazione che trova il nodo u che massimizza il rapporto viene fatta in "**find_max_difference(signed_edges)**". Successivamente, rimuoviamo tutti gli archi che hanno come sorgente o destinazione il nodo u, in maniera tale che il grado positivo e negativo di tutti i nodi vicini di u sia decrementato di 1.

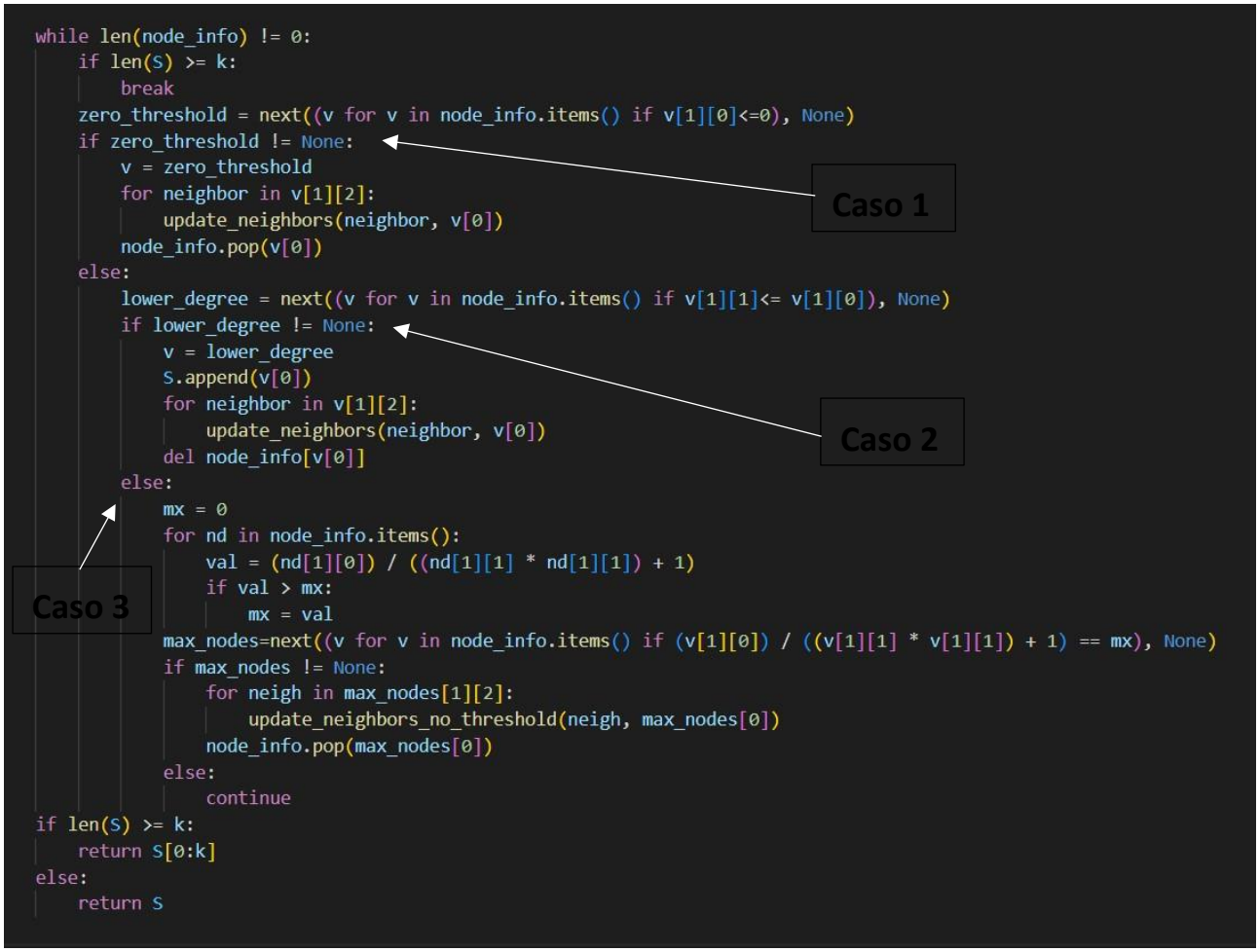
```
while len(S) < k:
        u = find_max_difference(signed_edges)
        signed_edges = [edge for edge in signed_edges if edge[0] != u or edge[1] != u]
        S.append(u)
return S
```

Figura 3-3-1 Algoritmo "Seeds-Greedy-Difference max"

3.3.2 Target-Set-Selection (G,k) – "TSS Algorithm"

Il secondo algoritmo implementato è l'algoritmo "Target-Set-Selection". Esso viene ripetuto finché non vengono rimossi tutti i nodi dal grafo, passando attraverso 3 casi principali:

I. Se esiste un nodo all'interno del grafo che ha threshold = 0, lo si rimuove e si aggiornano tutti i valori (threshold, grado e vicini) dei suoi vicini attraverso la funzione **update_neighbors(…)**
II. Altrimenti, se esiste un nodo all'interno del grafo che ha grado < threshold, lo si aggiunge ad S, lo si rimuove dal grafo e si aggiornano tutti i valori (threshold, grado e vicini) dei suoi vicini attraverso la funzione update_neighbors(…)
III. Altrimenti, si cerca il nodo all'interno del grafo che massimizza il rapporto **t(u) / (d(u) (d(u) + 1))**, lo si rimuove dal grafo e si aggiornano solo i valori di grado ed il numero dei vicini di esso.




```
while len(node_info) != 0:
    if len(S) >= k:
        break
    zero_threshold = next((v for v in node_info.items() if v[1][0]<=0), None)
    if zero_threshold != None:
        v = zero_threshold
        for neighbor in v[1][2]:
            update_neighbors(neighbor, v[0])
        node_info.pop(v[0])
    else:
        lower_degree = next((v for v in node_info.items() if v[1][1]<= v[1][0]), None)
        if lower_degree != None:
            v = lower_degree
            S.append(v[0])
            for neighbor in v[1][2]:
                update_neighbors(neighbor, v[0])
            del node_info[v[0]]
        else:
            mx = 0
            for nd in node_info.items():
                val = (nd[1][0]) / ((nd[1][1] * nd[1][1]) + 1)
                if val > mx:
                    mx = val
            max_nodes=next((v for v in node_info.items() if (v[1][0]) / ((v[1][1] * v[1][1]) + 1) == mx), None)
            if max_nodes != None:
                for neigh in max_nodes[1][2]:
                    update_neighbors_no_threshold(neigh, max_nodes[0])
                node_info.pop(max_nodes[0])
            else:
                continue
if len(S) >= k:
    return S[0:k]
else:
    return S
```

Figura 3-4 Algoritmo TSS

3.3.3 Algoritmo Seed Set Community Detection – "SSCD Algorithm"

Il terzo algoritmo implementato è quello ideato da noi. L'idea alla base dell'algoritmo è nata da un'osservazione fatta sul grafo di partenza, in particolare abbiamo notato come all'interno di quest'ultimo fossero presenti diverse comunità densamente connesse al loro interno. Di conseguenza l'algoritmo va a considerare tutte le comunità presenti nel grafo ed inserisce nel Seed Set il nodo di massimo grado positivo all'interno di ogni comunità. Questa operazione viene iterata fino a quando non sono stati inseriti tutti i k elementi all'interno del Seed Set.

Algorithm SS_CD (G,k)

```
1: S = ∅
2: while |S| < k do
3:     for each cmnty ∈ G(V, E) do
4:         w ← argmax_{w∈cmnty} d+(w)
5:         d(N(w)) = d(N(w)) − 1
6:         S = S ∪ {w}
7:         if |S| = k then
8:             break
9:         end if
10:    end for
11: end while
12: return S
```

Figura 3-5 Pseudocodice algoritmo SS_CD

L'implementazione in Python è stata effettuata attraverso l'uso di diverse variabili:

- **CmtyV**: contiene la lista delle comunità e delle modularità di ognuna di esse. Viene calcolato sfruttando il metodo CommunityCNM() di SNAP, che applica l'algoritmo di community detection di Clauset-Newman-Moore

```
def SS_CD(graph,signed_edges,k):
    CmtyV = graph.CommunityCNM()
```

Figura 3-6 Metodo di community detection fornito da SNAP

- **Cntr**: è un *dict* che contiene, per ogni nodo del grafo, il suo grado
- **Node_info**: è un *dict* che contiene, per ogni nodo, la lista dei suoi vicini

```
while len(S) < k:
    for Cmty in CmtyV[1]:
        mx = 0
        node = 0
        for NI in Cmty:
            val = cntr[NI]
            if val > mx:
                mx = val
                node = NI
        for neighbor in node_info[node]:
            for n in neighbor:
                cntr[n] -= 1
        S.append(node)
        if len(S) == k:
            break

return S
```

Figura 3-7 Algoritmo SS_CD

3.4 Funzione di cascade

La funzione di cascade viene utilizzata per influenzare tutta la rete a partire da un determinato Seed Set di nodi, identificato utilizzando uno degli algoritmi visti in precedenza. In particolare, per ogni nodo nel seed set, si controlla se la differenza tra il numero di archi positivi ed il numero di archi negativi uscenti è maggiore del threshold di un nodo che si trova all'esterno del seed set. Se è così, allora si aggiunge il nodo al set. In qualsiasi caso, l'algoritmo continua finché la lunghezza della lista contenente i nodi influenzati rimane la stessa anche nello step successivo (in pratica, non si riescono ad influenzare più nodi).

Nell'implementazione mostrata di seguito, facciamo uso di diverse strutture per portare a termine il lavoro. In particolare:

- **Influenced_nodes**: è la lista contenente tutti i nodi influenzati allo step r.
- **Previous_influenced_nodes**: è la lista contenente tutti i nodi influenzati allo step r-1

- **Difference_nodes**: contiene, per ogni nodo presente nella lista dei nodi influenzati, il valore della differenza tra il numero degli archi positivi e degli archi negativi di ogni nodo vicino ad esso e che si trova all'esterno del set.

```
def cascade_function(seed_set, threshold, signed_edges):
    influenced_nodes = []
    previous_influenced_nodes = []

    influenced_nodes = copy.deepcopy(seed_set)

    while len(influenced_nodes) != len(previous_influenced_nodes):
        previous_influenced_nodes = copy.deepcopy(influenced_nodes)
        difference_nodes = find_neighbors_in_seedset(signed_edges, influenced_nodes)
        for node,value in difference_nodes.items():
            if value>=threshold:
                influenced_nodes.append(node)

    return influenced_nodes
```

Figura 3-8 Funzione di cascade

4. I risultati ottenuti

I risultati mostrati in questa sezione mostrano per ogni fissata distribuzione di probabilità p,

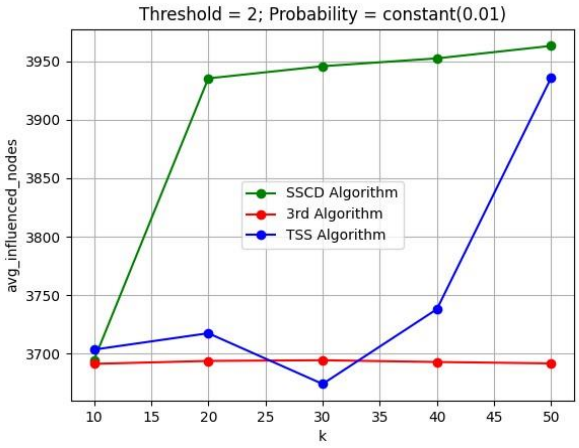
- per ciascuno dei tre algoritmi scelti (Seeds Greedy difference-max, TSS, SS_CD)
- al variare della size k del seed set
- i valori medi σ ottenuti

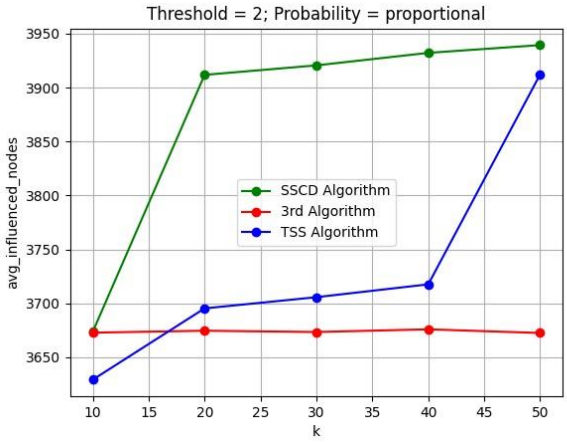
Per mostrare un risultato quanto più ampio ed affidabile possibile sono stati individuati diversi test case facendo variare di volta in volta la threshold e la probabilità con cui si effettua il labeling della signed network, mentre il valore di k variava da 10 a 50, con incrementi di 10, per tutti i test effettuati. In particolare, i test effettuati sono:

I. **Threshold** = 2 - - - - - - **Probabilità** = costante (pari a 0.01) II. **Threshold** = 2 - - - - - - **Probabilità** = 1/max{d(u),d(v)}
III. **Threshold** = 3 - - - - - - **Probabilità** = costante (pari a 0.01) IV. **Threshold** = 3 - - - - - - **Probabilità** = 1/max{d(u),d(v)}

4.1 Threshold = 2 - - - Probabilità = costante && Threshold = 2 - - - Probabilità = inversamente proporzionale al massimo grado di u e v

Osservando i grafici generati dai seguenti test case, si può da subito notare la differenza in termini di diffusione tra i tre algoritmi, in particolare tra l'SS_CD e gli altri due, sia nel test effettuato con probabilità costante, sia in quello con probabilità inversamente proporzionale al massimo grado dei nodi all'interno del grafo. Questa differenza diventa sempre più marcata con l'aumento graduale della dimensione k del Seed Set iniziale. L'algoritmo TSS in particolar modo riesce ad incrementare le proprie performance in termini di diffusione nel grafo a partire da un valore k in poi.





T = 2, **probabilità** = costante

Algoritmo	K=10	K=20	K=30	K=40	K=50
SS_CD	3673.6	**3935.4**	**3945.8**	**3952.5**	**3963.2**
TSS	**3703.3**	3717.4	3674.0	3738.0	3936.1
Difference-max	3691.4	3693.7	3694.3	3692.8	3691.6

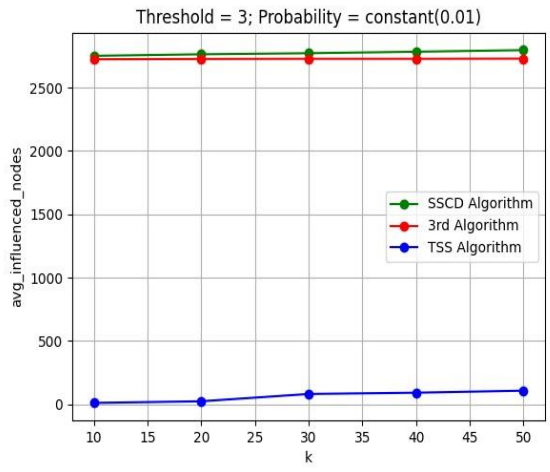
T = 2, **probabilità** = proporzionale

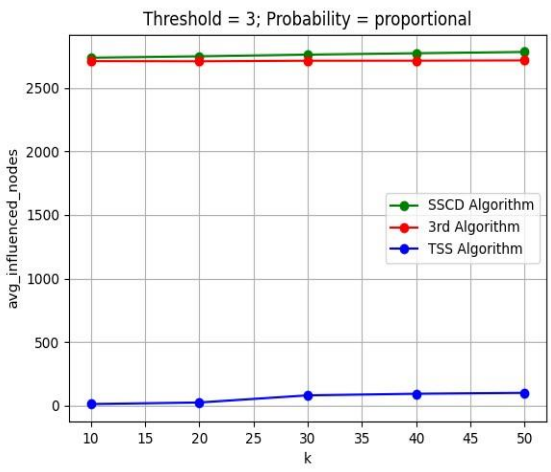
Algoritmo	K=10	K=20	K=30	K=40	K=50
SS_CD	**3694.5**	**3911.8**	**3920.6**	**3932.2**	**3939.4**

TSS	3629.2	3695.3	3705.7	3717.6	3912.2
Difference-max	3672.8	3674.6	3673.4	3675.9	3672.5

4.2 Threshold = 3 - - - Probabilità = costante && Threshold = 3 - - - Probabilità = inversamente proporzionale al massimo grado di u e v

Osservando i grafici generati dai seguenti test case, si può subito notare come l'algoritmo TSS fatichi ad influenzare i nodi del grafo, in quanto è molto probabile che il valore di threshold alto non gli consenta di diffondere l'influenza a partire dal seed set da lui trovato. Gli altri due algoritmi invece, riescono fin da subito ad influenzare una grande quantità di nodi, variando di poco all'aumentare di K. Per tutti i valori di K, l'algoritmo SS_CD continua a dimostrarsi superiore rispetto agli altri, influenzando una leggera quantità di nodi in più.





T = 3, **probabilità** = costante

Algoritmo	K=10	K=20	K=30	K=40	K=50
SS_CD	**2751.2**	**2763.5**	**2772.0**	**2783.7**	**2796.4**
TSS	11.0	23.9	81.6	91.6	107.8
Difference-max	2723.9	2726.5	2727.8	2727.7	2729.1

T = 3, **probabilità** = proporzionale

Algoritmo	K=10	K=20	K=30	K=40	K=50
SS_CD	**2738.7**	**2749.6**	**2762.7**	**2773.7**	**2783.8**
TSS	11.0	23.3	79.6	92.2	99.2
Difference-max	2711.8	2710.4	2715.0	2715.4	2717.6

5. Conclusioni

Dai risultati ottenuti nei test effettuati, abbiamo potuto osservare la bontà dell'algoritmo da noi proposto: **SS_CD**. In alcuni test, è riuscito a generare un seed set che ci ha permesso di influenzare la quasi totalità della rete, distaccando di parecchio sia l'algoritmo TSS, che quello greedy, soprattutto per valori di k molto bassi. Questo perché SS_CD riesce ad usufruire delle

caratteristiche della rete per ottenere un vantaggio rispetto agli altri algoritmi, che invece risultano molto generici e completamente ignari della struttura della rete.

Considerare la topologia della rete è fondamentale per fornire un algoritmo che possa sfruttarla al meglio. Proprio per questo, riteniamo che il nostro algoritmo sia particolarmente utile in tutte le reti in cui si vengono a presentare diverse **comunità**, ed in cui tutti i nodi all'interno di esse sono **fortemente connessi tra loro**, in quanto considera il nodo con grado positivo maggiore per ognuna di esse. Grazie a questa tecnica, infatti, il nodo considerato dall'algoritmo effettuerà la diffusione dell'influenza all'interno della propria comunità, facendo sì che tutte le comunità vengano influenzate in maniera indipendente.

Riteniamo inoltre che l'algoritmo possa essere **ulteriormente raffinato**, prendendo in considerazione, ad esempio, alcune modifiche alla ricerca dei nodi di ogni comunità da inserire nel seed set, che non necessariamente devono essere quelli con grado positivo maggiore. Nonostante il suo stato ancora embrionale, l'algoritmo ha avuto un notevole successo, per cui siamo ansiosi di vedere come si evolverà in futuro.