Report Progetto Algorithm Engineering

Giovanni Sicchio – 291326

Indice

Introduzione

Lo studio della centralità di un nodo all'interno di un grafo è un argomento cruciale in molti ambiti, in quanto fornisce una misura dell'importanza, dell'influenza o della posizione strategica di un nodo all'interno di una rete.

Con l'avvento dei Social Networks, gli studi relativi alla centralità dei nodi si sono intensificati, poiché un social network può essere rappresentato come un grafo in cui ogni soggetto rappresenta un nodo, la relazione tra due coppie di individui è rappresentata da un arco e la "forza" di tale relazione è rappresentata dal peso dell'arco. In questo contesto, la centralità indica quanto un individuo è posizionato centralmente in una rete sociale.

Nella network analysis si utilizzano principalmente quattro misure di centralità: *degree centrality, betweenness centrality, closeness centrality, eigenvector centrality*. In questo studio è stata utilizzata la *closeness centrality* (o centralità di vicinanza), ovvero, l'inverso della media dei cammini minimi da un vertice verso tutti gli altri vertici presenti nel grafo. Può anche essere vista come l'efficienza di un vertice di trasmettere informazioni a tutti gli altri vertici nel grafo.

Gli studi sul come calcolare nella maniera più efficiente le centralità dei vertici si basano sugli algoritmi di ranking, i quali mirano a classificare i vertici secondo un determinato criterio.

Descrizione del problema e motivazioni dello studio

La *closeness centrality* di tutti i vertici di un grafo può essere calcolata in molteplici modi.

Il calcolo più semplice prevede di risolvere il problema *All-Pairs-Shortest-Path,* il quale può essere risolto da vari algoritmi in un tempo $\boldsymbol{O(nm + n^2 log\, n)}$ dove *n* è il numero di vertici e *m* il numero di archi nel grafo. Questo approccio fornisce soluzioni corrette, ma non è efficiente per grafi di grandi dimensioni, come quelli con milioni di vertici.

Nell'articolo [1], che è alla base di questo studio, vengono definiti due algoritmi in grado di classificare i primi k vertici con maggiore centralità di vicinanza. Il primo è un algoritmo base noto come **TOPRANK(k)**. Il secondo algoritmo invece è una evoluzione del primo tramite l'aggiunta di una euristica e prende il nome di **TOPRANK2(k)**.

In questo articolo sono presenti considerazioni teoriche che dimostrano come entrambi gli algoritmi abbiano un tempo di esecuzione minore rispetto al metodo esatto e hanno un'alta probabilità di restituire un risultato corretto. Viene inoltre dimostrato, anche qui a livello teorico come, l'aggiunta di un'euristica migliori le prestazioni del secondo algoritmo rispetto al primo.

Stato dell'arte

Definizioni preliminari

I grafi utilizzati da questi algoritmi di ranking sono connessi, non direzionati e pesati *G(V,E)* con *n* vertici e *m* archi *(|V|=n, |E|=m)*, dove d(v,u) rappresenta la lunghezza di un cammino minimo da *v* a *u* e con Δ si denota il diametro del grafo, ovvero $\Delta = max\{v, u\ in\ V\}\, d(v,u)$.

La ***closeness centrality*** $\mathbf{c_v}$ di un vertice *v* è definita come INSERIRE FORMULA, in altre parole, è l'inverso della distanza media (dei cammini minimi) da *v* verso ogni altro vertice. Maggiore è $\mathbf{c_v}$, minore è la distanza media.

L'algoritmo esatto itera l'algoritmo di Dijkstra SSSP (Single Source Shortest Path) n volte per tutti gli n vertici e può essere efficientemente eseguito in $\boldsymbol{O(n\, log\, n + m)}$ per un singolo vertice.

RAND

RAND è un algoritmo di approssimazione sul quale si basano sia Toprank che Toprank2. Tale algoritmo è descritto in [2].

Questo algoritmo consente di stimare la centralità di tutti i vertici del grafo. Estraendo casualmente un certo numero di vertici e risolvendo per ognuno di loro il problema SSSP si otterranno dei risultati tramite cui si può stimare la centralità. Di seguito è riportato lo pseudocodice di questo algoritmo.

1. Let k be the number of iterations needed to obtain the desired error bound.
2. In iteration i, pick vertex v_i uniformly at random from G and solve the SSSP problem with v_i as the source.
3. Let
$$\hat{c}_u = \frac{1}{\sum_{i=1}^{k} \frac{n\, d(v_i,u)}{k(n-1)}}$$
be the centrality estimator for vertex u.

1. Nel primo step si definisce il numero di iterazioni necessarie per raggiungere un certo *error bound* ($\mathbf{Pr}\left\{\left|\frac{1}{\widehat{c_v}} - \frac{1}{c_v}\right| \geq \in \Delta\right\} \leq \frac{2}{n^{2l\frac{\in^2}{logn}(\frac{n-1}{n})^2}}$). Il numero di iterazioni corrisponde anche al numero di vertici che devono essere estratti casualmente dal grafo.
2. Nel secondo step di RAND, viene descritto cosa accade in una singola iterazione. Viene estratto un vertice in maniera casuale dal grafo e viene risolto il problema SSSP (utilizzando DIjkstra) ponendo il vertice estratto come sorgente.
3. Dopo aver eseguito le iterazioni, con i risultati ottenuti, si definisce uno stimatore di centralità utilizzando la formula descritta nello pseudocodice, ottenendo quindi delle stime sulla centralità di ogni altro vertice.

Utilizzando un numero di campioni pari a $l = \alpha\frac{logn}{\in^2}$ (con $\propto > 1$ e costante), allora l'algoritmo di approssimazione calcola le distanze medie dei cammini minimi dei vertici in $\boldsymbol{O}(\frac{\boldsymbol{logn}}{\in^2}(\boldsymbol{nlogn} + \boldsymbol{m}))$, con un errore additivo di $\in * \Delta$ e probabilità di successo almeno pari a $1 - \frac{1}{n}$ (con alta probabilità).

Toprank(k)

L'idea alla base di questo algoritmo è di utilizzare un approccio "misto" tra l'algoritmo di approssimazione RAND e l'algoritmo esatto.

Nella prima fase si utilizza l'algoritmo di approssimazione con *l* campioni per stimare le distanze medie di tutti i vertici. Successivamente, viene calcolato un insieme di vertici candidati *E,* che contiene i *top-k'* vertici con minore distanza media stimata. Dopo aver definito *E*, si va ad utilizzare l'algoritmo esatto per computare le distanze medie stimate esatte per ogni vertice in *E* ed ordinarle sulla base della centralità in modo da ottenere i primi k vertici.

La chiave di questo algoritmo è trovare il giusto bilanciamento tra il campione *l* e i candidati del set *k'* poiché, un valore di *l* troppo piccolo creerebbe un set *k'* troppo grande e viceversa. Idealmente si cerca un valore *l* che minimizza la somma *l* + *k'*.

Di seguito è riportato lo pseudocodice dell'algoritmo Toprank.

Algorithm ***TOPRANK(k)***

1 Use the approximation algorithm RAND with a set S of ℓ sampled vertices to obtain the estimated average distance $\hat{a}_v$ for each vertex v.
// Rename all vertices to $\hat{v}_1, \hat{v}_2, \ldots, \hat{v}_n$ such that $\hat{a}_{\hat{v}_1} \leq \hat{a}_{\hat{v}_2} \leq \cdots \leq \hat{a}_{\hat{v}_n}$.
2 Find $\hat{v}_k$.
3 Let $\hat{\Delta} = 2 \min_{u \in S} \max_{v \in V} d(u, v)$.
// $d(u, v)$ for all $u \in S, v \in V$ have been calculated at step 1 and $\hat{\Delta}$ is determined in $O(\ell n)$ time.
4 Compute candidate set E as the set of vertices whose estimated average distances are less than or equal to $\hat{a}_{\hat{v}_k} + 2f(\ell) \cdot \hat{\Delta}$.
5 Calculate exact average shortest-path distances of all vertices in E.
6 Sort the exact average distances and find the top-k vertices as the output.

1. Nella prima fase di esecuzione dell'algoritmo si va ad utilizzare l'algoritmo di approssimazione RAND, il quale estrarrà un set *S* di *l* campioni, per stimare le distanze medie per ogni vertice. Sulla base di queste distanze stimate, i vertici vengono ordinati in ordine crescente in base alle medie e successivamente rinominati (*v1, v2, ..., vn*).
2. Si identifica il vertice *vk* nell'insieme dei vertici ordinati nello step precedente.
3. Si definisce il valore di Δ utilizzando la formula descritta nello pseudocodice, tenendo conto che i vertici *u* fanno parte dell'insieme dei vertici estratti nello step 1 e *v* all'insieme di tutti i vertici del grafo.
4. Si calcola la soglia limite (formula nello pseudocodice) per identificare i vertici che faranno parte dell'insieme di candidati *E*. Questi vertici avranno una distanza media stimata minore o uguale alla soglia.

   NOTA: come specificato in [1] la funzione utilizzata è $\boldsymbol{f(l) = \alpha' \sqrt{\frac{logn}{l}}}$
5. Si utilizza l'algoritmo esatto (Dijkstra) per calcolare le distanze medie esatte per tutti i vertici presenti in E.
6. Si ordinano i vertici in base alle distanze medie esatte (calcolate in 5) e si estraggono i *top k* vertici in output.

Secondo gli autori di [1] l'algoritmo è in grado di classificare i vertici correttamente indipendentemente dal valore del parametro *l*.

Per quanto riguarda le prestazioni di questo algoritmo, la maggior parte del tempo di esecuzione è speso nella risoluzione degli *l* problemi SSSP (Step 1) e degli |*E*| problemi SSSP (Step 5). Per poter minimizzare i tempi, è necessario minimizzare la somma *l*+|*E*| agendo sul termine *l* (dato che la dimensione di *E* è dipendente da tale valore).

La scelta ottimale per *l* è: $\boldsymbol{\theta(n^{\frac{2}{3}} log^{\frac{1}{3}} n)}$. Scegliendo questo valore, il tempo di esecuzione totale dell'algoritmo è pari a $\boldsymbol{O((k + n^{\frac{2}{3}} log^{\frac{1}{3}} n)(nlogn + m))}$

Dato che l'algoritmo esatto impiegherebbe un tempo pari a $\boldsymbol{O(nm + n^2 log\, n)}$, si ha un miglioramento delle prestazioni ma tale miglioramento non è così significativo. Da qui, la scelta degli autori di sviluppare l'algoritmo Toprank2.

Toprank2(k)

Come detto nella sezione precedente, il miglioramento del tempo di esecuzione ottenuto con Toprank non è molto significativo. L'idea base di Toprank2 è quella di realizzare un algoritmo di ranking eseguibile in un tempo $\boldsymbol{O(poly(k)(nlogn + m))}$ con *poly(k)* che è una funzione polinomiale dipendente da k. In questo modo il numero di SSSP da eseguire non sarà più dipendente da *k* e *n* ma solo da *k*.

In questo algoritmo è stato scelto un approccio euristico. L'euristica è stata realizzata in modo da aggiungere incrementalmente nuovi campioni per computare distanze medie più accurate per tutti i vertici. In ogni iterazione vengono aggiunti *q* nuovi vertici e, dopo aver computato le nuove distanze medie, si otterrà un nuovo set di vertici candidati *E*. Se la dimensione del nuovo set di candidati diminuisce più di *q*, allora è chiaro che i risparmi ottenuti dalla riduzione del numero di candidati superano il costo dell'aggiunta di altri campioni. In questo caso si procede con l'iterazione successiva aggiungendo altri campioni.

Di seguito lo pseudocodice dell'algoritmo e quindi dell'implementazione dell'euristica. I primi quattro step sono identici al caso Toprank.

Algorithm ***TOPRANK2(k)***

1 Use the approximation algorithm RAND with a set S of ℓ sampled vertices to obtain the estimated average distance $\hat{a}_v$ for each vertex v.
// Rename all vertices to $\hat{v}_1, \hat{v}_2, \ldots, \hat{v}_n$ such that $\hat{a}_{\hat{v}_1} \leq \hat{a}_{\hat{v}_2} \leq \cdots \leq \hat{a}_{\hat{v}_n}$.
2 Find $\hat{v}_k$.
3 Let $\hat{\Delta} = 2 \min_{u \in S} \max_{v \in V} d(u, v)$.
4 Compute candidate set E as the set of vertices whose estimated average distances are less than or equal to $\hat{a}_{\hat{v}_k} + 2f(\ell) \cdot \hat{\Delta}$.
5 **repeat**
6 $p \leftarrow |E|$
7 Select additional q vertices S^+ as new samples uniformly at random.
8 Update estimated average distances of all vertices using new samples in S^+ (need to compute SSSP for all new sample vertices).
9 $S \leftarrow S \cup S^+; \ell \leftarrow \ell + q; \hat{\Delta} \leftarrow \min(\hat{\Delta}, 2 \min_{u \in S^+} \max_{v \in V} d(u, v))$
// Rename all vertices to $\hat{v}_1, \hat{v}_2, \ldots, \hat{v}_n$ such that $\hat{a}_{\hat{v}_1} \leq \hat{a}_{\hat{v}_2} \leq \cdots \leq \hat{a}_{\hat{v}_n}$.
10 Find $\hat{v}_k$.
11 Compute candidate set E as the set of vertices whose estimated average distances are less than or equal to $\hat{a}_{\hat{v}_k} + 2f(\ell) \cdot \hat{\Delta}$.
12 $p' \leftarrow |E|$
13 **until** $p - p' \leq q$
14 Calculate exact average shortest-path distances of all vertices in E.
15 Sort the exact average distances and find the top-k vertices as the output.

---

1. Come in Toprank, nella prima fase di esecuzione dell'algoritmo si va ad utilizzare l'algoritmo di approssimazione RAND, il quale estrarrà un set *S* di *l* campioni, per stimare le distanze medie per ogni vertice. Sulla base di queste distanze stimate, i vertici vengono ordinati in ordine crescente in base alle medie e successivamente rinominati (*v1, v2, ..., vn*).
2. Si identifica il vertice *vk* nell'insieme dei vertici ordinati nello step precedente.
3. Si definisce il valore di Δ utilizzando la formula descritta nello pseudocodice, tenendo conto che i vertici *u* fanno parte dell'insieme dei vertici estratti nello step 1 e *v* all'insieme di tutti i vertici del grafo.
4. In questo step di calcola la soglia limite per definire quali vertici faranno parte dell'insieme dei candidati (formula nello pseudocodice). Tutti i vertici, con distanza media stimata minore o uguale alla soglia, saranno inseriti all'interno dell'insieme di candidati *E*.

   NOTA: come specificato in [1] la funzione utilizzata è $\boldsymbol{f(l) = \alpha' \sqrt{\frac{logn}{l}}}$
5. In questo passo inizia il ciclo e quindi l'implementazione dell'euristica con i relativi passi da ripetere.
6. Alla variabile *p* viene assegnato il valore della dimensione dell'insieme dei vertici campione *E*.
7. Si selezionano *q* vertici aggiuntivi come nuovo campione random (*S+*).
8. Si aggiornano le distanze medie stimate calcolate precedentemente per tutti i vertici usando i vertici del nuovo campione. In questa fase si andrà quindi ad eseguire un SSSP per ogni vertice del nuovo campione.

9. In questo step si aggiornano le variabili e gli insiemi esistenti. L'insieme dei vertici estratti random viene aggiornato e definito come unione dei vertici precedentemente estratti e quelli estratti nello step 7. Il valore di *l* (numero di campioni estratti) viene aggiornato come somma del numero di campioni estratti precedentemente e i *q* nuovi vertici. Viene calcolato il nuovo diametro stimato con la formula riportata nello pseudocodice (usando quindi quanto calcolato nello step 8). Successivamente si effettua, come nello step 1, una fase di ordinamento e rinominazione dei vertici.
10. Si identifica il nuovo vertice *vk* nell'insieme dei vertici ordinati nello step precedente.
11. Come nello step 4, si calcola la soglia limite per definire quali vertici faranno parte del nuovo insieme dei candidati. Tutti i vertici, con distanza media stimata minore o uguale alla soglia, saranno inseriti all'interno dell'insieme di candidati *E*.
12. Alla variabile *p'* viene assegnato il valore della dimensione del nuovo insieme *E*.
13. Si verifica se la differenza tra le variabili *p* e *p'* è minore o uguale al numero di nuovi vertici estratti. Se la differenza è minore o uguale si esce dal ciclo altrimenti si ripete lo pseudocodice ripartendo dallo step 6.
14. Si utilizza l'algoritmo esatto (Dijkstra) per calcolare le distanze medie esatte per tutti i vertici presenti in E.
15. Si ordinano i vertici in base alle distanze medie esatte (calcolate in 5) e si estraggono i *top k* vertici in output.

Essenzialmente, questa nuova versione di Toprank consente di trovare dinamicamente il valore ottimale di *l* per ottimizzare la somma *l*+|*E*|.

Il valore di *l*, per gli stessi ragionamenti effettuati nella sezione dedicata all'algoritmo Toprank, è pari a $\boldsymbol{\theta}(\boldsymbol{n^{\frac{2}{3}}\, log^{\frac{1}{3}} n})$. Per quanto riguarda il valore di *q* invece, gli autori di [1], affermano che si può utilizzare un valore piccolo, come ad esempio $\boldsymbol{log n}$.

In [1] non sono specificati tempi di esecuzione per questo algoritmo, nemmeno a livello teorico. Viene semplicemente affermato che l'utilizzo di questo algoritmo dovrebbe fornire prestazioni migliori ma che andrebbero effettuati studi più approfonditi per determinare gli errori presenti nelle soluzioni calcolate.

Obiettivo dello studio

In questo studio l'obiettivo è dimostrare, tramite l'implementazione dei due algoritmi, se effettivamente impiegano meno tempo rispetto al metodo esatto e se l'algoritmo euristico è migliore di quello base. Verranno quindi effettuati dei test per verificare il tempo di esecuzione dei due algoritmi e la qualità delle soluzioni calcolate.

Valutazione sperimentale

Ambiente di test

Il codice per l'implementazione dell'algoritmo e per l'esecuzione degli esprimenti è disponibile su GitHub al seguente link.

La cartella è organizzata nel seguente modo:

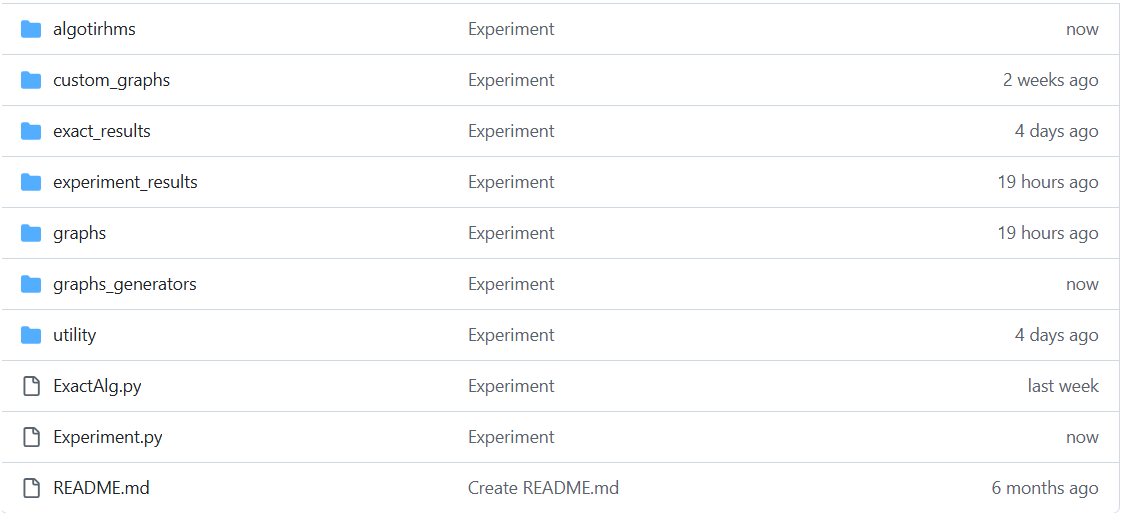

Fig. 1: Struttura cartella Github

- La cartella *algorithms* contiene le implementazioni Python degli algoritmi Toprank e Toprank2.
- La cartella *custom_graphs* contiene dei grafi generati da un generatore custom. Tali grafi sono stati utilizzati in fase di test per la realizzazione dei due algoritmi.
- La cartella *exact_results* contiene i risultati delle esecuzioni dell'algoritmo esatto (implementato nel file *ExactAlg.py*) utilizzando i grafi contenuti della directory *graphs*.
- La cartella *graphs* contiene i grafi utilizzati nell'esperimento vero e proprio. Questi grafi sono stati realizzati con un generatore specifico presente nella cartella *graphs_generators*.
- La cartella *graphs_generators* contiene due script python, uno per ogni generatore. Un generatore è custom (custom_graphs_generator), mentre l'altro generatore (graph_generator) è un generatore di grafi Barabási-Albert per reti sociali. Entrambi i generatori realizzano grafi connessi, orientati e pesati.
- La cartella *utility* contiene due file python. Il file *RAND.py* contiene l'implementazione dell'algoritmo RAND descritto in [2]. Il file *utils.py* contiene invece le funzioni relative agli step comuni agli algoritmi Toprank e Toprank2.
- Il file *ExactAlg.py* contiene l'implementazione dell'algoritmo esatto per la risoluzione del problema del ranking basato sulla centralità di vicinanza.
- Il file *Experiment.py* contiene l'implementazione dell'esperimento.

Si rimanda al file *README.md*, mostrato in figura (Fig. 1) e presente sulla cartella GitHub, per la configurazione dell'ambiente locale e per l'istallazione delle librerie necessarie per l'esecuzione corretta dell'esperimento.

Implementazione degli algoritmi

Gli algoritmi di ranking, come detto precedentemente, sono stati implementati in Python e le loro implementazioni sono contenute nella sottocartella *algorithms* presente nella cartella Github. Si è scelto di utilizzare Python poiché, data la sua semplicità, ha consentito di eseguire i test più velocemente. Inoltre, per gli obiettivi posti, non è stato necessario utilizzare un linguaggio più efficiente per la valutazione delle prestazioni poiché i risultati raggiungibili tramite Python sono sufficienti.

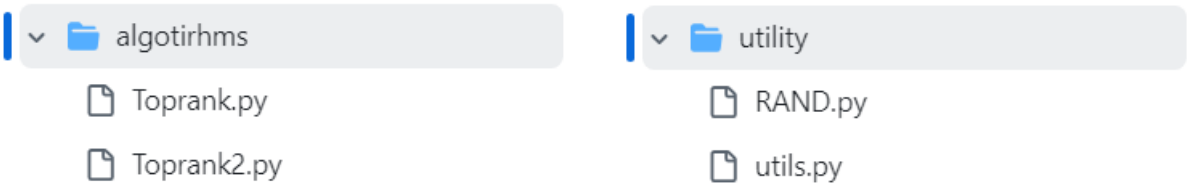

Fig 2: Cartelle degli algoritmi e delle utilities

Come detto precedentemente, nella cartella *algorithms* sono presenti i file Python con le implementazioni dei singoli algoritmi (*Toprank.py* e *Toprank2.py*). Nella cartella *utility* invece, ci sono due file, uno con l'implementazione dell'algoritmi RAND e l'altro (*utils.py*) che contiene una serie di funzioni necessarie per l'implementazione dei due algoritmi di ranking.

Nello specifico, il file *utils.py*, contiene le funzioni necessarie per:

- indentificare l'insieme dei vertici candidati ad essere nelle prime k posizioni;
- calcolare le distanze esatte per un set di vertici;
- estrarre i primi k vertici in base alla centralità di vicinanza.

Sia il file *Toprank.py* che *Toprank2.py* hanno al proprio interno un main eseguibile. Questi main consentono di poter testare entrambi gli algoritmi su un singolo grafo (modificando il codice del main con il nome del grafo da analizzare). Per entrambi questi esempi non è previsto il salvataggio dei risultati poiché sono solo script di test.

In entrambi gli algoritmi, si è scelto di seguire quanto definito in [1] per l'assegnazione di alcuni valori:

- $\boldsymbol{l = n^{\frac{2}{3}} log^{\frac{1}{3}} n}$. Numero di campioni da estrarre nello Step 1 di entrambi gli algoritmi (RAND)
- $\boldsymbol{f(l) = \alpha' \sqrt{\frac{logn}{l}}}$. Funzione necessaria per il calcolo della soglia per l'inserimento dei vertici nel set dei candidati.
- $\boldsymbol{\alpha' > 1}$. Coefficiente della funzione $\boldsymbol{f(l)}$, scelto con il valore di **1.25** per evitare di calcolare una soglia troppo alta, che quindi consenta a tutti i vertici di entrare nell'insieme dei candidati per il caso *Toprank*.

Generazione dei grafi e dei risultati esatti

Per le fasi di test e sperimentazione, si è scelto di utilizzare grafi generati tramite dei generatori Python, poiché nelle principali fonti online non sono disponibili grafi con caratteristiche compatibili con i requisiti dei due algoritmi di ranking.
Come descritto nelle sezioni precedenti, i grafi da fornire in input ai due algoritmi devono essere connessi, orientati e pesati.

La generazione dei grafi è stata affidata ai due generatori contenuti nella directory *graphs_generators* della cartella GitHub.

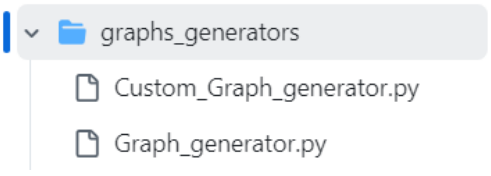

Fig 3: Cartella dei generatori di grafi

Per ottenere dei risultati esatti da utilizzare negli esperimenti, è stato implementato un algoritmo esatto per classificare i vertici di un grafo in base della loro centralità di vicinanza. Questo algoritmo è presente nella cartella GitHub (directory principale), nel file python *ExactAlg.py*.

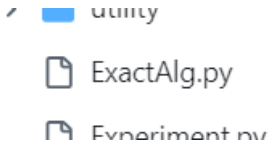

Fig 4: File algoritmo esatto

Generatore custom

Per la generazione dei grafi nella fase di test iniziale, è stato utilizzato lo script *Custom_Graph_generator.py*. Questo script consente di creare grafi con le caratteristiche richieste ma senza alcuna proprietà tipica delle reti sociali. Per tale motivo, è stato impiegato esclusivamente nella fase preliminare di test per verificare la corretta implementazione dei due algoritmi.

Nello specifico, il generatore è basato sulla seguente funzione:

```python
def create_connected_weighted_graph(num_nodes):
    # Grafo vuoto
    G = nx.Graph()

    G.add_nodes_from(range(num_nodes))

    # Creazione dell'albero per assicurare che il grafo sia connesso
    for i in range(num_nodes - 1):
        weight = random.randint( a: 1,  b: 11)
        G.add_edge(i, i + 1, weight=weight)

    # Aggiunta ulteriori archi casuali per aumentare la densità del grafo
    additional_edges = random.randint(num_nodes, 2 * num_nodes)
    while additional_edges > 0:
        u = random.randint( a: 0, num_nodes - 1)
        v = random.randint( a: 0, num_nodes - 1)
        if u != v and not G.has_edge(u, v):  # Evita duplicati
            weight = random.randint( a: 1,  b: 10)
            G.add_edge(u, v, weight=weight)
            additional_edges -= 1

    return G
```

Fig 5: Funzione di generazione di grafi custom

La funzione prende come parametro il numero di nodi da inserire all'interno del grafo. Dopo aver aggiunto tutti i nodi, la funzione crea un albero per garantire che il grafo sia connesso. Successivamente, vengono aggiunti altri archi causali, in un numero compreso tra il numero di nodi e il suo doppio, evitando duplicati.

Personalizzando i valori presenti all'interno del *main* è possibile generare più grafi e salvarli automaticamente all'interno della directory *custom_graphs*.

Generatore di Barabási-Albert

Per gli esperimenti, sono stati scelti grafi più realistici, rappresentativi delle reti sociali. Si è utilizzata una versione modificata di un generatore di Barabási-Albert, implementata nel file *Graph_generator.py*.

Di base, tale generatore crea grafi non orientati e non pesati. Per ottenere grafi con le caratteristiche richieste, è stata utilizzata la seguente funzione:

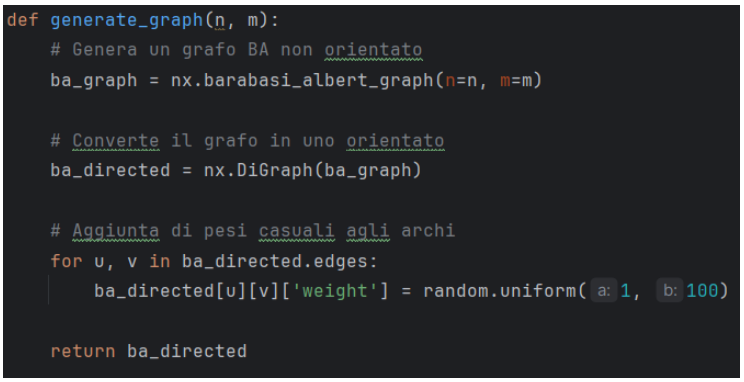

Fig 6: Funzione di generazione dei grafi

La funzione accetta come parametri il numero di nodi e il numero di connessioni che ogni nuovo nodo aggiunge ai nodi esistenti. Il valore di *m* deve essere minore nel numero totale di nodi e maggiore di 1. Per simulare una rete più realistica, è stato scelto *m=5*.
Il processo include la generazione di un grafo non orientato, la sua conversione in un grafo orientato e l'assegnazione di pesi causali ad ogni singolo arco.

Anche in questo caso, è possibile modificare i parametri nel file main per generare grafi personalizzati, salvati automaticamente nella directory *graphs*.

Algoritmo esatto

Lo scirpt *ExactAlg.py* implementa una versione esatta degli algoritmi di classificazione.

Inizialmente, l'algoritmo recupera un grafo (definito nel *main*) e calcola le distanze esatte di ogni vertice rispetto a tutti gli altri utilizzando l'algoritmo di Dijkstra.

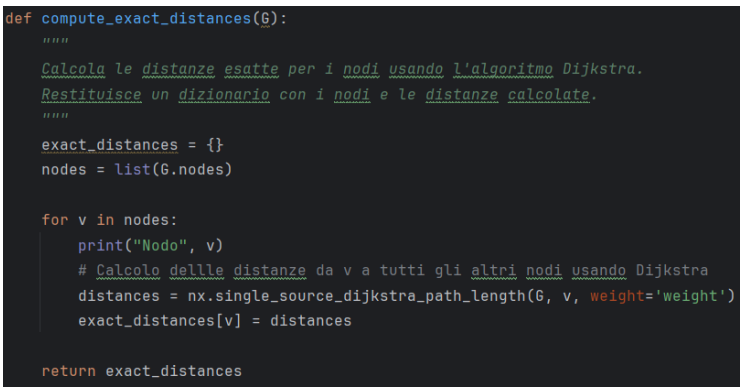

Fig 7: Funzione di computazione delle distanze esatte

Successivamente, sulla base delle distanze calcolate, vengono determinate le centralità di vicinanza e la distanza media per ciascun vertice.
Infine, i vertici si ordinano in ordine decrescente di centralità di vicinanza (o distanza media crescente), e i risultati sono salvati in un file *centrality_results_n* (n viene sostituito con il numero di nodi del grafo) nella directory *exact_results*.

Questo algoritmo non è incluso direttamente negli esperimenti, poiché la sua esecuzione su grafi di grandi dimensioni aumenterebbe significativamente la durata complessiva. I valori esatti vengono quindi calcolati preliminarmente per poi confrontarli con i risultati degli esperimenti.

Definizione degli esperimenti

Per eseguire gli esperimenti, è stato sviluppato lo script Python *Experiment.py*, disponibile nella directory principale del repository GitHub. Questo script definisce tutti i passaggi per eseguire i due algoritmi di classificazione, variando i grafi e il numero di vertici da estrarre per le prime *k* posizioni. Dopo le esecuzioni si determina l'andamento della durata degli algoritmi e della qualità dei risultati e si generano dei grafici e delle tabelle.

Nella prima parte del *main*, vengono definiti i valori di *k* da testare e inizializzate le strutture dati per i risultati intermedi. Successivamente, i grafi vengono recuperati della cartella *graphs*.

```
k_value = [3, 5, 10, 15, 20, 50]
differences = []
results = []
durations = []
# Step 1: import dei grafi
graphs = []
# Recupero e ordinamento dei file per numero di nodi
graph_files = [filename for filename in os.listdir("graphs") if filename.endswith(".pkl")]
graph_files_sorted = sorted(graph_files, key=lambda x: int(x.split('_')[1].split('.')[0]))

# Caricamento i grafi in ordine
for filename in graph_files_sorted:
    file_path = os.path.join("graphs", filename)
    with open(file_path, "rb") as f:
        G = pickle.load(f)
        graphs.append(G)
```

Fig 8: Fase di inizializzazione degli esperimenti

Un ciclo iterativo varia i valori di *k,* mentre un ciclo interno itera sui grafi caricati. Durante ogni iterazione, vengono eseguiti i due algoritmi (*Toprank* e *Toprank2)* per ciascun valore di *k* e grafo *G*.

```
# Step 2: inizio ciclo (per ogni k e per ogni grafo)
for k in k_value:
    for graph in graphs:

        # Step 2.1 applicazione degli algoritmi di ranking
        print(f"Test su grafo con n = {graph.number_of_nodes()} e m = {graph.number_of_edges()}")

        # Step 2.1.1: esegui Toprank per grafo g, salva risultato e durata
        print(f"Inizio esecuzione Toprank su grafo con n = {graph.number_of_nodes()}")
        start = datetime.now()
        res_1 = Toprank(graph, k)
        end = datetime.now()
        dur_1 = end - start
        for i, (vertex, avg_distance) in enumerate(res_1, start=1):
            print(f"v{i}: Nodo originale: {vertex}, Distanza media esatta: {avg_distance}")
        print(f"Durata dell'esecuzione di Toprank: {dur_1}")

        # Step 2.1.2: esegui Toprank2 per grafo g, salva risultato e durata
        print(f"Inizio esecuzione Toprank 2 su grafo con n = {graph.number_of_nodes()}")
        start = datetime.now()
        res_2 = Toprank2(graph, k)
        end = datetime.now()
        dur_2 = end - start
        for i, (vertex, avg_distance) in enumerate(res_2, start=1):
            print(f"v{i}: Nodo originale: {vertex}, Distanza media esatta: {avg_distance}")
        print(f"Durata dell'esecuzione di Toprank 2: {dur_2}")

        results.append((k, res_1, res_2))
        durations.append((k, graph.number_of_nodes(), dur_1.total_seconds(), dur_2.total_seconds()))
```

Fig 9: Iterazione base di un esperimento

Prima di passare al successivo grafo o valore di *k*, si confrontano le soluzioni ottenute da entrambi gli algoritmi con la soluzione esatta, valutando le differenze di posizione tra i nodi. Il calcolo. Ad esempio: nella soluzione esatta il nodo 24 si trova in posizione 2 mentre nella soluzione di uno dei due algoritmi di classificazione si trova in posizione 4, in questo caso la differenza è 2.

La differenza totale tra l'algoritmo esatto e un algoritmo di classificazione è data dalle somme delle singole differenze di posizione.

```
# Step 2.2: confronta soluzione 1 e 2 con esatto e determina differenza, salva differenza
diff_1 = calculate_position_difference(res_1, graph)
print(f"Somma delle differenze per Toprank: {diff_1}")

diff_2 = calculate_position_difference(res_2, graph)
print(f"Somma delle differenze per Toprank 2: {diff_2}")

differences.append((k, graph.number_of_nodes(), diff_1, diff_2))
```

Fig 10: Chiamate funzioni per calcolo delle differenze

```
def calculate_position_difference(result, graph):
    # Nome del file da caricare
    filename = f"centrality_results_{graph.number_of_nodes()}.txt"
    filepath = os.path.join("exact_results", filename)

    # Lettura del file di centralità e conversione in una lista ordinata
    centrality_positions = {}
    with open(filepath, "r") as file:
        # Salta l'intestazione
        next(file)
        for position, line in enumerate(file, start=1):
            parts = line.split()  # Divide per spazi o tab
            vertex = int(parts[0])  # Primo valore: Nodo
            centrality_positions[vertex] = position

    # Calcola la differenza di posizione
    total_difference = 0
    for rank, (vertex, _) in enumerate(result, start=1):
        if vertex in centrality_positions:
            file_position = centrality_positions[vertex]
            total_difference += abs(rank - file_position)
        else:
            print(f"Vertex {vertex} non trovato nel file centralità.")

    return total_difference
```

Fig 11: Funzione per il calcolo delle posizioni di differenza

Al termine delle iterazioni, lo script genera grafici e tabelle, tra cui:

- Un grafico per ogni k, che mostra la variazione del tempo di esecuzione al variare del numero di nodi.

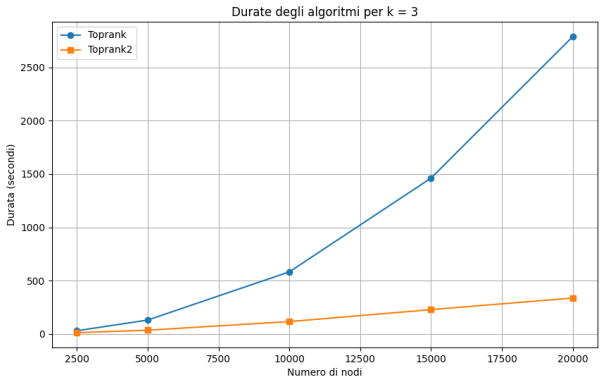

Fig 12: Grafico tempo di esecuzione con k=3

- Un grafico per ogni k, che illustra la differenza tra le soluzioni calcolate e quelle esatte.

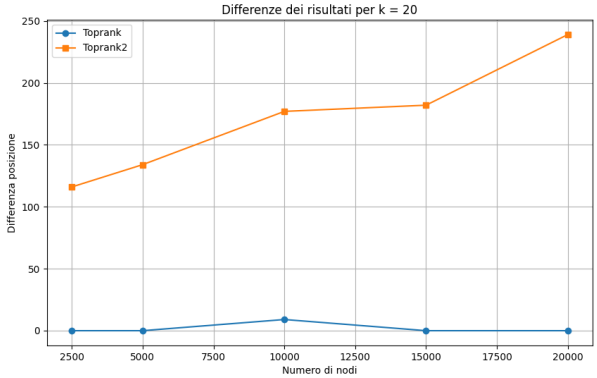

Fig 13: Grafico differenze soluzioni con k=20

- Tabelle riassuntive per ogni valore di k.

Numero di nodi	Durata Toprank (s)	Durata Toprank2 (s)	Differenza Toprank	Differenza Toprank2
2500.0	30.226169	12.045866	0.0	0.0
5000.0	130.073913	35.269255	0.0	1.0
10000.0	582.295589	115.978192	0.0	2.0
15000.0	1461.037371	228.015929	0.0	0.0
20000.0	2787.688143	336.676636	0.0	0.0

Fig14: Tabella riassuntiva per k=3

- Un grafico per ogni grafo, ognuno dei quali riporta i tempi di esecuzione al variare di k.

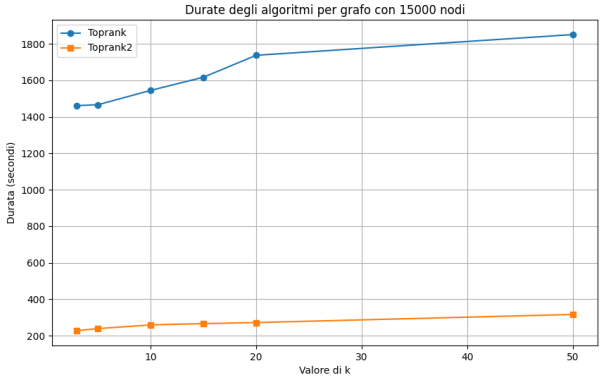

Fig 15: Grafico tempi di esecuzione per 15.000 nodi

- Un grafico per ogni grafo con le differenze tra soluzione esatta e calcolata.

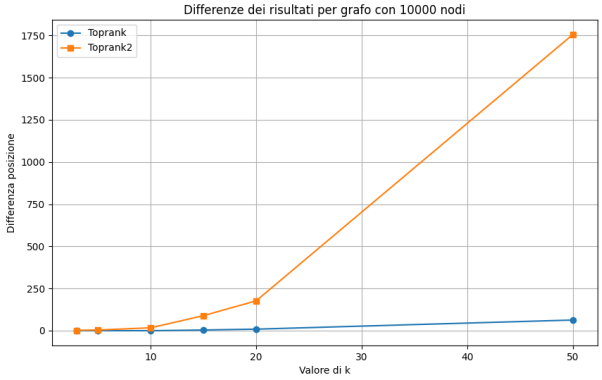

Fig 16: Grafico differenze soluzioni per 10.000 nodi

- Una tabella riassuntiva per ogni grafo.

Valore di k	Durata Toprank (s)	Durata Toprank2 (s)	Differenza Toprank	Differenza Toprank2
3.0	30.226169	12.045866	0.0	0.0
5.0	29.341878	12.317455	0.0	1.0
10.0	30.130046	12.469004	0.0	18.0
15.0	30.270519	13.199468	0.0	52.0
20.0	30.297502	12.556027	0.0	116.0
50.0	30.098281	14.828143	0.0	583.0

Fig 17: Tabella riassuntiva per grafo con 2.500 nodi

Per maggiori dettagli sulla configurazione e l'esecuzione degli esperimenti, consultare il *README* presente nella cartella GitHub.

Analisi dei risultati

In questa sezione verranno analizzati i dati ottenuti dall'esecuzione di un esperimento. Come descritto in precedenza, i risultati esatti di un algoritmo di classificazione basato sulla centralità di vicinanza possono essere calcolati tramite lo script *ExactAlg.py*, illustrato nella sezione dedicata. Questi risultati forniscono una base per valutare la correttezza delle soluzioni generate dagli algoritmi implementati.

L'esperimento ha coinvolto cinque grafi di dimensione crescenti, generati dal generatore di Barabási-Albert, composti rispettivamente da:

- 2.500 nodi
- 5.000 nodi
- 10.000 nodi
- 15.000 nodi
- 20.000 nodi

Per ciascun grafo, la soluzione esatta è stata calcolata e salvata nella cartella *exact_results*.

Sono stati testati i seguenti valori di *k*: 3, 5, 10, 15, 20 e 50 al fine di valutare come questo parametro influisca sulle prestazioni degli algoritmi e sulla qualità delle soluzioni prodotte.

L'esecuzione dello script *Experiment.py*, della durata di circa dieci ore e mezza, ha generato i grafici e le tabelle utili per analizzare due aspetti fondamentali:

- Il tempo di esecuzione degli algoritmi.
- La qualità delle soluzioni rispetto a quelle esatte.

Nelle sezioni successive, questi risultati verranno discussi in dettaglio, focalizzandosi su prestazioni e accuratezza.

Analisi del tempo di esecuzione

L'analisi del tempo di esecuzione dei due algoritmi, *Toprank* e *Toprank2*, è stata condotta considerando due diverse prospettive:

- Variazione del numero di nodi del grafo con *k* fisso.
- Variazione del valore di *k* con numero di nodi fisso.

1. Variazione del numero di nodi con *k* fisso

Quando il valore di *k* è mantenuto fisso e il numero dei nodi del grafo aumenta, il tempo di esecuzione cresce per entrambi gli algoritmi. Tuttavia, gli andamenti mostrano differenza significative:

- *Toprank* presenta un incremento esponenziale del tempo di esecuzione all'aumentare del numero di nodi.
- *Toprank2*, invece, ha un tempo di esecuzione che cresce in modo più lineare.

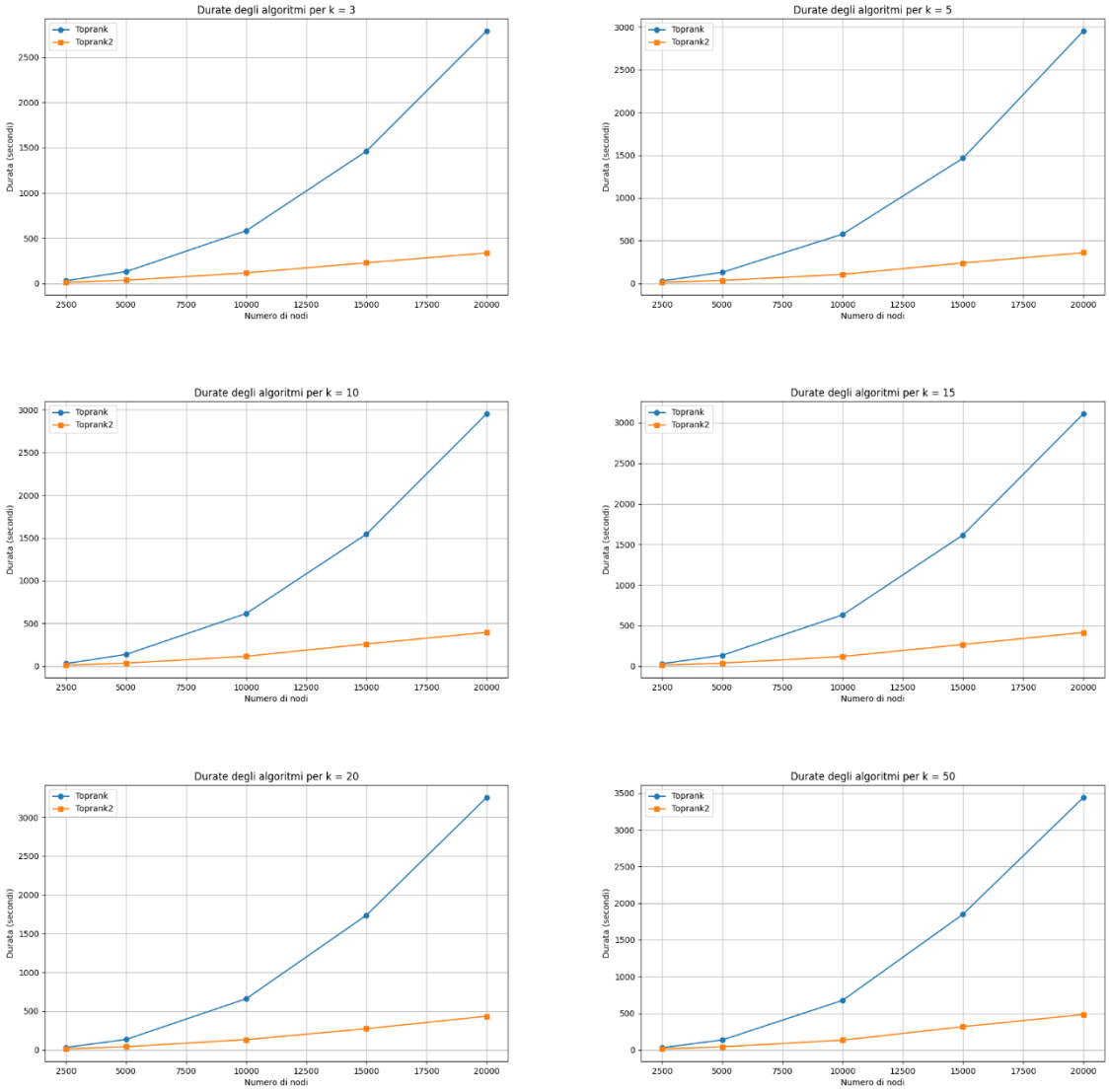

Fig 18: Grafici sulla durata degli algoritmi con k fisso

Un 'analisi dettagliata dei valori evidenzia che:

- Per ogni valore di *k,* il tempo di esecuzione di *Toprank* aumenta di quattro/cinque al raddoppiare nel numero di vertici.
- Per *Toprank2,* il tempo di esecuzione cresce di circa tre volte al raddoppio del numero di vertici.

Numero di nodi	Durata Toprank (s)	Durata Toprank2 (s)	Differenza Toprank	Differenza Toprank2
2500.0	30.226169	12.045866	0.0	0.0
5000.0	130.073913	35.269255	0.0	1.0
10000.0	582.295589	115.978192	0.0	2.0
15000.0	1461.037371	228.015929	0.0	0.0
20000.0	2787.688143	336.676636	0.0	0.0

Fig 19: Tabella riassuntiva per k=3

Numero di nodi	Durata Toprank (s)	Durata Toprank2 (s)	Differenza Toprank	Differenza Toprank2
2500.0	29.341878	12.317455	0.0	1.0
5000.0	129.631468	34.639439	0.0	3.0
10000.0	576.973487	105.848234	0.0	4.0
15000.0	1465.821253	239.439199	0.0	1.0
20000.0	2954.30532	359.557662	0.0	0.0

Fig 20: Tabella riassuntiva per k=5

Numero di nodi	Durata Toprank (s)	Durata Toprank2 (s)	Differenza Toprank	Differenza Toprank2
2500.0	30.130046	12.469004	0.0	18.0
5000.0	137.159155	35.291213	0.0	26.0
10000.0	616.272032	115.344299	0.0	17.0
15000.0	1544.512339	259.509485	0.0	28.0
20000.0	2952.778874	395.177446	0.0	16.0

Fig 21: Tabella riassuntiva per k=10

Numero di nodi	Durata Toprank (s)	Durata Toprank2 (s)	Differenza Toprank	Differenza Toprank2
2500.0	30.270519	13.199468	0.0	52.0
5000.0	133.834015	37.48744	0.0	63.0
10000.0	631.944627	119.05639	4.0	89.0
15000.0	1616.59095	266.835038	5.0	94.0
20000.0	3109.544527	414.753485	0.0	96.0

Fig 22: Tabella riassuntiva per k=15

Numero di nodi	Durata Toprank (s)	Durata Toprank2 (s)	Differenza Toprank	Differenza Toprank2
2500.0	30.297502	12.556027	0.0	116.0
5000.0	133.347502	40.107466	0.0	134.0
10000.0	659.255855	131.284509	9.0	177.0
15000.0	1736.711217	272.379418	0.0	182.0
20000.0	3255.648311	433.345329	0.0	239.0

Fig 23: Tabella riassuntiva per k=20

Numero di nodi	Durata Toprank (s)	Durata Toprank2 (s)	Differenza Toprank	Differenza Toprank2
2500.0	30.098281	14.828143	0.0	583.0
5000.0	135.956757	42.586868	22.0	1724.0
10000.0	675.835862	133.212394	63.0	1755.0
15000.0	1850.277429	316.8398	14.0	1867.0
20000.0	3442.067106	482.477101	0.0	2023.0

Fig 24: Tabella riassuntiva per k=50

2. Variazione del valore di *k* con numero di nodi fisso

Quando il numero di nodi è fisso e si varia il valore di *k,* per un grafo con 2.500 nodi, il tempo di esecuzione di *Toprank* rimane quasi costante al variare di *k*. Per *Toprank2*, invece, si osserva un andamento crescente con una diminuzione del tempo tra k=15 e k=20.

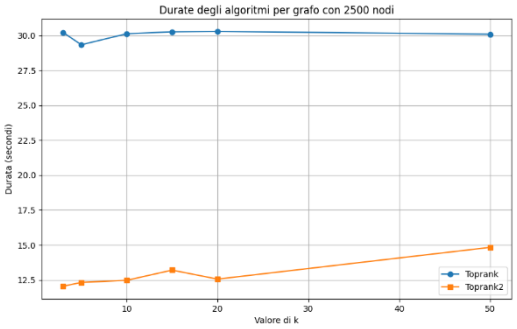

Fig 25: Grafico tempo di esecuzione per grafo con 2.500 nodi

Per un grafo con 5.000 nodi, il comportamento è simile: *Toprank2* si stabilizza, mentre *Toprank* presenta una leggera fase decrescente sempre negli stessi valori del caso precedente.

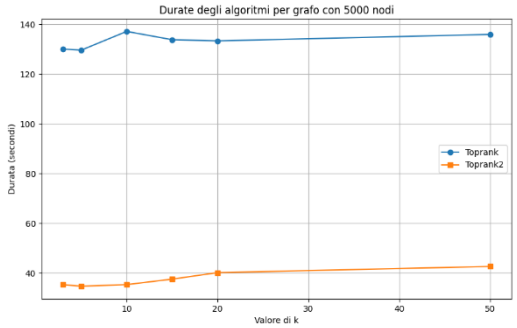

Fig 26: Grafico tempo di esecuzione per grafo con 5.000 nodi

Questi andamenti decrescenti possono essere attribuiti alla fase iniziale degli algoritmi, basata su un'estrazione casuale di campioni. Per grafi di dimensioni ridotte, il basso numero di vertici estratti potrebbe influire sull'andamento.

Per grafi con 10.000, 15.000 e 20.000 nodi, il tempo di esecuzione aumenta in modo uniforme al crescere di *k*. Un''unica eccezione si verifica per il grafo con 10.000 nodi e *k* pari a 5, dove un basso numero di vertici estratti potrebbe aver influenzato il risultato.

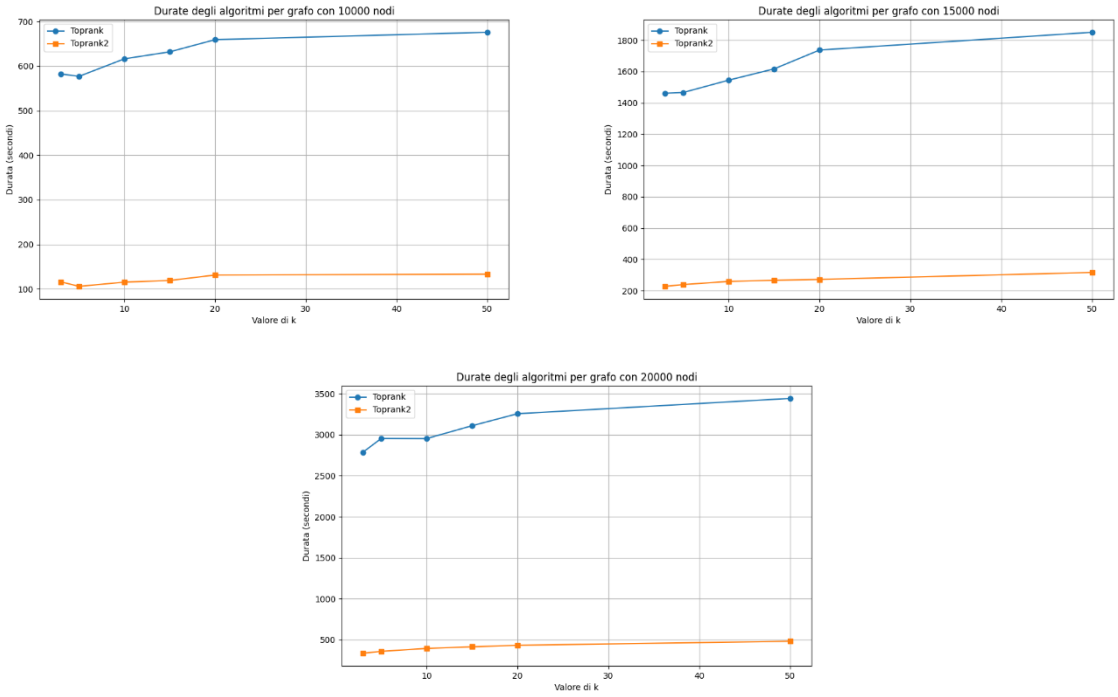

Fig 25: Grafico tempo di esecuzione per grafi con 10.000, 15.000 e 20.000 nodi

Un'analisi maggiormente dettagliata mostra che:

- Per l'algoritmo *Toprank,* il tempo di esecuzione rimane quasi costante (circa 30 secondi) per il grafo con 2.500 nodi. Per i grafi più grandi, il tempo di esecuzione cresce con un incremento inferiore al 10% al raddoppio di *k*.
- Per *Toprank2,* si osserva un andamento crescente simile, con un aumento inferiore 10% rispetto al raddoppio del numero di campioni estratti.

Valore di k	Durata Toprank (s)	Durata Toprank2 (s)	Differenza Toprank	Differenza Toprank2
3.0	30.226169	12.045866	0.0	0.0
5.0	29.341878	12.317455	0.0	1.0
10.0	30.130046	12.469004	0.0	18.0
15.0	30.270519	13.199468	0.0	52.0
20.0	30.297502	12.556027	0.0	116.0
50.0	30.098281	14.828143	0.0	583.0

Valore di k	Durata Toprank (s)	Durata Toprank2 (s)	Differenza Toprank	Differenza Toprank2
3.0	130.073913	35.269255	0.0	1.0
5.0	129.631468	34.639439	0.0	3.0
10.0	137.159155	35.291213	0.0	26.0
15.0	133.834015	37.48744	0.0	63.0
20.0	133.347502	40.107466	0.0	134.0
50.0	135.956757	42.586868	22.0	1724.0

Fig 26: Tabelle riassuntive per grafi con 2.500 e 5.000 nodi

Valore di k	Durata Toprank (s)	Durata Toprank2 (s)	Differenza Toprank	Differenza Toprank2
3.0	582.295589	115.978192	0.0	2.0
5.0	576.973487	105.848234	0.0	4.0
10.0	616.272032	115.344299	0.0	17.0
15.0	631.944627	119.05639	4.0	89.0
20.0	659.255855	131.284509	9.0	177.0
50.0	675.835862	133.212394	63.0	1755.0

Valore di k	Durata Toprank (s)	Durata Toprank2 (s)	Differenza Toprank	Differenza Toprank2
3.0	1461.037371	228.015929	0.0	0.0
5.0	1465.821253	239.439199	0.0	1.0
10.0	1544.512339	259.509485	0.0	28.0
15.0	1616.59095	266.835038	5.0	94.0
20.0	1736.711217	272.379418	0.0	182.0
50.0	1850.277429	316.8398	14.0	1867.0

Valore di k	Durata Toprank (s)	Durata Toprank2 (s)	Differenza Toprank	Differenza Toprank2
3.0	2787.688143	336.676636	0.0	0.0
5.0	2954.30532	359.557662	0.0	0.0
10.0	2952.778874	395.177446	0.0	16.0
15.0	3109.544527	414.753485	0.0	96.0
20.0	3255.648311	433.345329	0.0	239.0
50.0	3442.067106	482.477101	0.0	2023.0

Fig 27: Tabella riassuntiva per grafi con 10.000, 15.000 e 20.000 nodi

Analisi della correttezza dei risultati

In questa sezione viene effettuata una analisi delle differenze tra le soluzioni calcolate dai due algoritmi (*Toprank* e *Toprank2*) e la soluzione esatta, attraverso due tipi di grafici:

- *k* fisso e numero di nodi variabile.
- Numero di nodi fisso e *k* variabile.

Si ricorda che il valore della differenza corrisponde alle differenze di posizioni tra i vertici presenti nella soluzione di uno dei due algoritmi e la soluzione esatta per un determinato grafo.

1. ***k* fisso, numero di nodi variabile**

   Quando k=3 e k=5, l'algoritmo *Toprank* mostra una differenza costante pari a zero, mentre per *Toprank2* la differenza cresce all'aumentare del numero di nodi fino a raggiungere un massimo per grafi da 10.000 nodi. Successivamente, la differenza diminuisce fino a tornare a zero per grafi di dimensioni maggiori (15.000 e 20.000 nodi). Questo andamento potrebbe dipendere dal basso numero di nodi combinato con un ridotto numero di vertici da estrarre, che potrebbe provocare una maggiore quantità di errori.

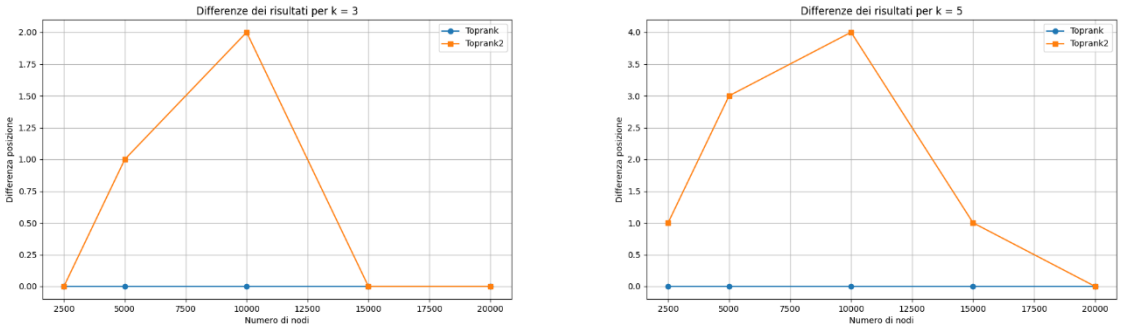

Fig 28: Grafici differenze dei risultati per k=3 e k=5

Nel caso con k=10, l'algoritmo *Toprank* continua a mantenere una differenza costante pari a zero. Per l'algoritmo *Toprank2,* invece, si osserva un andamento diverso: la differenza segue un andamento "sinusoidale", aumentando l'ampiezza delle sinusoidi all'aumentare della dimensione del grafo.

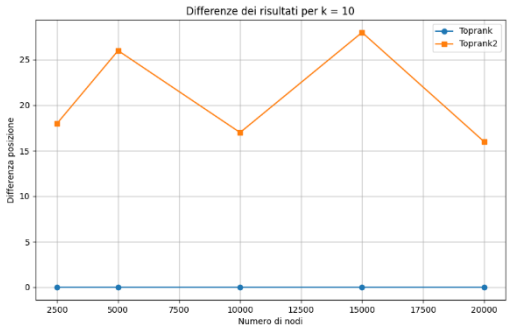

Fig 29: Grafico differenze dei risultati per k=10

Quando *k* assume valori più elevati (k= 5,20,50) l'algoritmo *Toprank* non mantiene più un valore costante pari a zero. Si osservano piccoli scostamenti per grafi da 5.000, 10.000 e 15.000 nodi, ma la differenza ritorna a zero per grafi di 20.000 nodi. L'algoritmo *Toprank2*, invece, mostra una differenza crescente all'aumentare del numero di nodi.

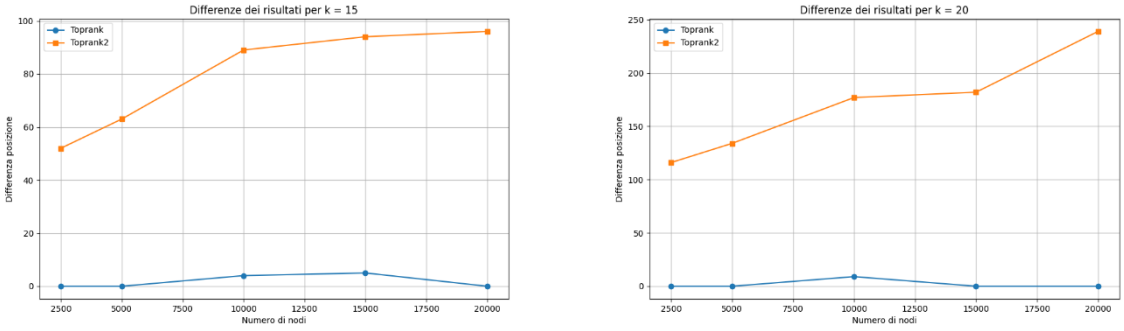

Fig 30: Grafici delle differenze dei risultati per k=15 e k=20

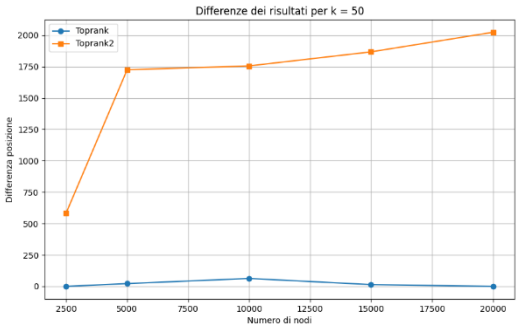

Fig 31: Grafico delle differenze dei risultati per k=50

Esaminando le tabelle riassuntive, sono confermati gli andamenti presentati nei grafici ma è possibile aggiungere maggiore dettaglio.

Per k=3 e k=5, la differenza per *Toprank* è sempre pari a zero, mentre *Toprank2* segue un andamento crescente fino ai grafi da 10.000 nodi e decresce nei grafi più grandi. In questo caso non si hanno maggiori dettagli rispetto l'analisi precedente.

Numero di nodi	Durata Toprank (s)	Durata Toprank2 (s	Differenza Toprank	Differenza Toprank2
2500.0	30.226169	12.045866	0.0	0.0
5000.0	130.073913	35.269255	0.0	1.0
10000.0	582.295589	115.978192	0.0	2.0
15000.0	1461.037371	228.015929	0.0	0.0
20000.0	2787.688143	336.676636	0.0	0.0

Numero di nodi	Durata Toprank (s)	Durata Toprank2 (s	Differenza Toprank	Differenza Toprank2
2500.0	29.341878	12.317455	0.0	1.0
5000.0	129.631468	34.639439	0.0	3.0
10000.0	576.973487	105.848234	0.0	4.0
15000.0	1465.821253	239.439199	0.0	1.0
20000.0	2954.30532	359.557662	0.0	0.0

Fig 32: Tabella riassuntiva per k=3 e k=5

Per k=10, è confermata una variazione "sinusoidale" nella differenza di *Toprank2* con l'altezza di queste sinusoidi che si amplifica all'aumentare del numero di nodi del grafo. L'aumento/diminuzione del valore della differenza è di circa il 40% al raddoppiare del numero di nodi. Per quanto riguarda *Toprank,* anche in questo caso il valore resta costante a zero.

Numero di nodi	Durata Toprank (s)	Durata Toprank2 (s	Differenza Toprank	Differenza Toprank2
2500.0	30.130046	12.469004	0.0	18.0
5000.0	137.159155	35.291213	0.0	26.0
10000.0	616.272032	115.344299	0.0	17.0
15000.0	1544.512339	259.509485	0.0	28.0
20000.0	2952.778874	395.177446	0.0	16.0

Fig 33: Tabella riassuntiva per k=10

Per k=15, k=20 e k=50, *Toprank* inizia a mostrare lievi scostamenti dai valori nulli, mentre *Toprank2* presenta una crescita marcata, con incrementi di circa il 20% al raddoppiare del numero di nodi. In particolare, per k=50, si osserva una triplicazione della differenza tra 2.500 e 5.000 nodi.

Numero di nodi	Durata Toprank (s)	Durata Toprank2 (s)	Differenza Toprank	Differenza Toprank2
2500.0	30.270519	13.199468	0.0	52.0
5000.0	133.834015	37.48744	0.0	63.0
10000.0	631.944627	119.05639	4.0	89.0
15000.0	1616.59095	266.835038	5.0	94.0
20000.0	3109.544527	414.753485	0.0	96.0

Numero di nodi	Durata Toprank (s)	Durata Toprank2 (s)	Differenza Toprank	Differenza Toprank2
2500.0	30.297502	12.556027	0.0	116.0
5000.0	133.347502	40.107466	0.0	134.0
10000.0	659.255855	131.284509	9.0	177.0
15000.0	1736.711217	272.379418	0.0	182.0
20000.0	3255.648311	433.345329	0.0	239.0

Numero di nodi	Durata Toprank (s)	Durata Toprank2 (s)	Differenza Toprank	Differenza Toprank2
2500.0	30.098281	14.828143	0.0	583.0
5000.0	135.956757	42.586868	22.0	1724.0
10000.0	675.835862	133.212394	63.0	1755.0
15000.0	1850.277429	316.8398	14.0	1867.0
20000.0	3442.067106	482.477101	0.0	2023.0

Fig 34: Tabelle riassuntive per k=15, k=20 e k=50

2. Numero di nodi fisso, k variabile

Analizzando la differenza tre le soluzioni mantenendo fisso il numero di nodi e variando il valore di *k,* per l'algoritmo *Toprank* si osserva che la differenza si allontana dallo zero solo per valori elevati di *k* (esclusi i grafi con 2.500 e 5.000 nodi, dove la differenza rimane costante pari a zero). Per l'algoritmo *Toprank2*, invece, si nota una crescita esponenziale all'aumentare del valore di *k*.

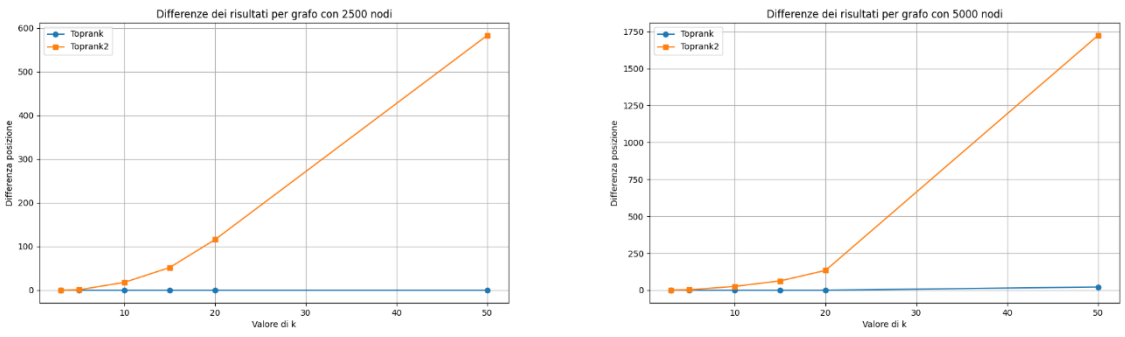

Fig 35: Grafici delle differenze dei risultati per 2.500 e 5.000 nodi

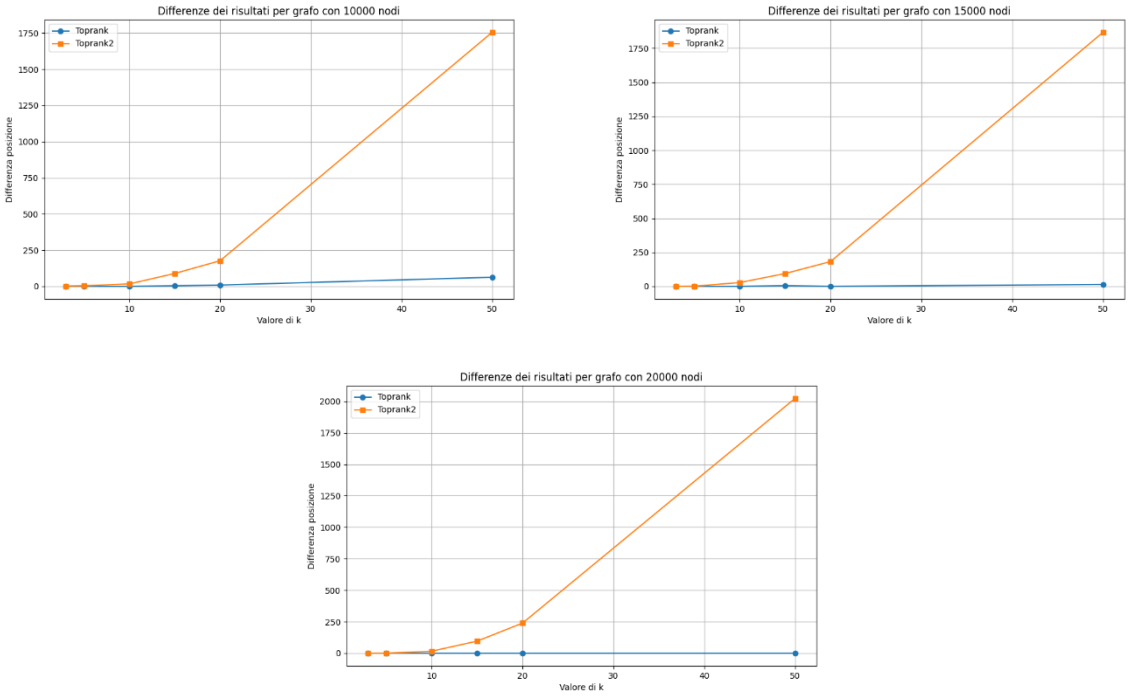

Fig 36: Grafici delle differenze dei risultati per 10.000, 15.000 e 20.000 nodi

Analizzando le tabelle riassuntive per i singoli grafi si può notare un andamento comune per quanto riguarda le differenze calcolate da *Toprank2* in cui, al raddoppiare del valore di *k,* la distanza tra le soluzioni aumenta di più del doppio. Per quanto riguarda *Toprank,* anche in questo caso nota un andamento crescente ma più irregolare e meno accentuato. Nello specifico:

- Per un grafo da 5.000 nodi, la differenza calcolata da *Toprank* passa da 0.0 a 22.0 quando *k* varia da 20 a 50.

Valore di k	Durata Toprank (s)	Durata Toprank2 (s)	Differenza Toprank	Differenza Toprank2
3.0	130.073913	35.269255	0.0	1.0
5.0	129.631468	34.639439	0.0	3.0
10.0	137.159155	35.291213	0.0	26.0
15.0	133.834015	37.48744	0.0	63.0
20.0	133.347502	40.107466	0.0	134.0
50.0	135.956757	42.586868	22.0	1724.0

Fig 37: Tabella riassuntiva per grafo con 5.000 nodi

- Per un grafo da 10.000 nodi, la differenza cresce da 4.0 (k=15) a 63.0 (k=50), confermando l'andamento esponenziale.

Valore di k	Durata Toprank (s)	Durata Toprank2 (s	Differenza Toprank	Differenza Toprank2
3.0	582.295589	115.978192	0.0	2.0
5.0	576.973487	105.848234	0.0	4.0
10.0	616.272032	115.344299	0.0	17.0
15.0	631.944627	119.05639	4.0	89.0
20.0	659.255855	131.284509	9.0	177.0
50.0	675.835862	133.212394	63.0	1755.0

Fig 38. Tabella riassuntiva per grafo con 10.000 nodi

- Per un grafo da 15.000 nodi, la differenza presenta un andamento simile a quello del grafo da 5.000 nodi, con un picco significativo per k=50.

Valore di k	Durata Toprank (s)	Durata Toprank2 (s	Differenza Toprank	Differenza Toprank2
3.0	1461.037371	228.015929	0.0	0.0
5.0	1465.821253	239.439199	0.0	1.0
10.0	1544.512339	259.509485	0.0	28.0
15.0	1616.59095	266.835038	5.0	94.0
20.0	1736.711217	272.379418	0.0	182.0
50.0	1850.277429	316.8398	14.0	1867.0

Fig 39: Tabella riassuntiva per grafo con 15.000 nodi

- Nei grafi da 2.500 e 20.000 nodi, invece, la differenza rimane costante pari a zero.

Valore di k	Durata Toprank (s)	Durata Toprank2 (s)	Differenza Toprank	Differenza Toprank2
3.0	30.226169	12.045866	0.0	0.0
5.0	29.341878	12.317455	0.0	1.0
10.0	30.130046	12.469004	0.0	18.0
15.0	30.270519	13.199468	0.0	52.0
20.0	30.297502	12.556027	0.0	116.0
50.0	30.098281	14.828143	0.0	583.0

Valore di k	Durata Toprank (s)	Durata Toprank2 (s	Differenza Toprank	Differenza Toprank2
3.0	2787.688143	336.676636	0.0	0.0
5.0	2954.30532	359.557662	0.0	0.0
10.0	2952.778874	395.177446	0.0	16.0
15.0	3109.544527	414.753485	0.0	96.0
20.0	3255.648311	433.345329	0.0	239.0
50.0	3442.067106	482.477101	0.0	2023.0

Fig 40: Tabelle riassuntive per grafi con 2.500 e 20.000 nodi

Conclusioni

In questo report sono stati valutati sperimentalmente gli algoritmi *Toprank* e *Toprank2*, con l'obiettivo di valutare i tempi di esecuzione e la correttezza delle soluzioni, al fine di dimostrare se l'algoritmo euristico *Toprank2* rappresenti un miglioramento rispetto a *Toprank*.

Entrambi gli algoritmi sono stati implementati in Python e testati sugli stessi grafi per garantire condizioni di confronto uniformi.

Dai risultati emerge che *Toprank* è costante e preciso nella maggior parte dei casi, con differenze rispetto alle soluzioni esatte generalmente nulle o trascurabili. Tuttavia, per valori elevati di *k* o grafi di dimensioni intermedie, sono stati osservati lievi scostamenti.

Al contrario, *Toprank2* ha mostrato una maggiore variabilità, influenzata dal numero di nodi del grafo e dal valore di *k*. Per bassi valori di *k,* le differenze seguono andamenti irregolari, legati alla natura casuale di alcune funzioni interne. Nei casi con valori alti di *k*, invece, la differenza con le soluzioni esatte cresce esponenzialmente, risultando più marcati rispetto a *Toprank*.

In sintesi, *Toprank* si rivela più stabile e adatto a contesti in cui è richiesta un'elevata precisione, a scapito dei tempi di esecuzione. *Toprank2,* invece, è più rapido nei calcoli, ma produce soluzioni di qualità inferiore, risultando quindi più adatto a scenari in cui la rapidità di calcolo è prioritaria e la tolleranza all'errore è maggiore.

Bibliografia

[1] Kazuya Okamoto, Wei Chen0, and Xiang-Yang Li - Ranking of Closeness Centrality for Large-Scale Social Networks - Springer-Verlag Berlin Heidelberg 2008 pp. 186–195. Link.

[2] Eppstein, D., Wang, J.: Fast Approximation of Centrality. Journal of Graph Algorithms and Applications 8, 39–45 (2004). Link.