UNIVERSITÀ DI ROMA TOR VERGATA

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Studio e sviluppo di strategie per la riduzione del random-walk controversy score tra echo chambers dei social networks

Relatore:
Giuseppe F. Italiano

Correlatore:
Nikos Parotsidis

Laureando:
Stefano Agostini
matricola: 0234240

A.A. 2017/2018

Da inserire qui tutte le dediche della mia tesi...

Sommario

Esistono, e sono sempre esistiti, particolari temi, detti *controversi*, per i quali ognuno di noi si schiera come sostenitore o come oppositore; tali temi possono riguardare contesti politici, sociali o culturali. L'effetto quasi immediato è la divisione della popolazione in due gruppi che hanno visioni opposte sull'argomento *controverso* in considerazione e che difficilmente scambiano tra loro informazioni e punti di vista, non facendo altro che acutizzare la loro inconciliabilità di opinione: gruppi di individui di questo genere sono anche detti *echo chambers*. Gli invidui facenti parte della stessa *echo chamber* hanno quindi le stesse credenze e lo stesso parere riguardo all'argomento *controverso*: essi rinforzano a vicenda le proprie opinioni e sono scarsamente esposti a punti di vista opposti ai propri (i.e. le opinioni che caratterizzano l'altra *echo chamber*). Non è immune da tale fenomeno il mondo dei *social media*. In particolare, *Twitter* identifica un particolare *topic* (i.e. argomento) mediante un *hashtag* (e.g. $\#novax$) e la discussione riguardo a tale *topic* può essere descritta mediante un *endorsement graph*, ovvero un *grafo diretto* in cui ciascun nodo rappresenta un utente che partecipa alla discussione e vi è un arco diretto da un nodo x ad un nodo y se e solo se l'utente x approva l'opinione contenuta in un *tweet* dell'utente y (dove l'approvazione è espressa per mezzo dello strumento del *retweet*). Quando il *topic* in considerazione è *controverso*, la struttura dell'*endorsement graph* mette in luce la presenza di gruppi (di utenti) molto connessi al loro interno ma che comunicano poco tra loro: la distanza di opinione che separa tali gruppi (o *echo-chambers*)

viene quantificata mediante il così detto *indice di controversia* del grafo, che viene misurato, nel lavoro di tesi proposto, utilizzando una metrica basata sui *random walks*. *Endorsement graphs* con elevati *indici di controversia* sono caratterizzati da *echo chambers* poco connesse tra loro, in ciascuna delle quali è amplificata una visione univoca ed acritica sull'argomento *controverso* considerato.
Con l'obiettivo di ridurre efficacemente tale *indice di controversia*, è stato implementato un *framework*, utilizzando il linguaggio *Python* e con l'ausilio della libreria *NetworkX*, il quale:

1. acquisisce i dati necessari per costruire l'*endorsement graph* associato ad un *hashtag controverso*, presente nel *social network* di *Twitter*, fornito in input. Cattura la collezione dei dati necessari mediante la libreria *Python Tweepy* che accede all'*API* di *Twitter*;

2. esegue un algoritmo che identifica le *echo chambers*;

3. implementa un *edge-recommendation system* che permette di individuare k archi diretti (k fornito in *input*) che, se aggiunti al grafo, riducono il suo *grado di controversia*. Nella pratica si vuole esporre alcuni utenti al contenuto di altri sperando che lo approvino mediante *retweet*: l'effetto di questa approvazione nell'*endorsement graph* sarebbe la comparsa degli archi consigliati;

4. offre un *tool* di visualizzazione degli archi individuati all'interno del grafo.

Con riferimento al punto *3*, il problema di ottimizzazione che si vorrebbe risolvere sarebbe quello di trovare il *set* di k archi diretti che, se aggiunti al grafo, minimizzano il *grado di controversia*. Poiché gli *endorsement graphs* dei *social networks* sono generalmente costituiti da un numero molto elevato di nodi (indicato con n), risolvere tale problema di ottimizzazione considerando tutte le possibili combinazioni degli

archi a gruppi di k (complessità $O(\binom{n^2}{k})$) (approccio *brute force*) è evidentemente molto costoso dal punto di vista computazionale e molto inefficiente anche per quanto riguarda i tempi di esecuzione. Pertanto l'approccio seguito è quello adottato nell'articolo *"Reducing controversy by connecting opposing views"*[4], che consiste nel considerare solo un sottoinsieme degli archi possibili (i.e. un sottoinsieme degli archi non ancora materializzati nell'*endorsement graph*) ed estrarre da questo sottoinsieme i k più promettenti. Chiaramente questa soluzione potrebbe restituire archi meno efficaci, per quanto riguarda la riduzione del *grado di controversia* che consentono, rispetto a quelli restituiti dall'approccio *brute force* ma apporta un miglioramento in termini di efficienza computazionale; in particolare, l'euristica che specifica la modalità di scelta del sottoinsieme degli archi candidati è cruciale. L'euristica utilizzata in questo lavoro di tesi è quella proposta nell'articolo [4], la quale consiste nel considerare solo gli archi diretti che permettono di connettere i vertici di grado alto della prima comunità (o *echo chamber*) con i vertici di grado alto della seconda comunità e viceversa: da questo sottoinsieme di archi vengono estratti i k più promettenti in termini di riduzione del *grado di controversia* che consentono.
La bontà di tale euristica è stata valutata considerando due algoritmi alternativi, che si occupano di estrarre i k archi più promettenti dal sottoinsieme considerato:

- *non-greedy*: vengono scelti in un solo *step* i k archi che porterebbero al *grado di controversia* più basso qualora venissero aggiunti al grafo *individualmente*;

- *greedy*: vengono scelti k archi in k *steps*, in ognuno dei quali viene estratto l'arco migliore, tra quelli rimanenti, in termini di decremento del *grado di controversia* che apporterebbe se fosse aggiunto al grafo.

Nel lavoro di tesi verranno descritte le modalità di implementazione di tali algoritmi e successivamente verranno confrontati tra loro in termini di efficacia (ossia in termini del decremento del *grado di controversia* che consentono, qualora tutti

gli archi che consigliano venissero accettati) ed in termini di tempi di esecuzione ed efficienza computazionale: a tal fine sono stati condotti *tests* su tre *endorsement graphs* di *Twitter* corrispondenti ad *hashtags* particolarmente controversi (*#beefban,#russia_march,#indiana*), volutamente scelti da contesti sociali e culturali diversi in modo tale da ottenere un'analisi più attendibile.

L'algoritmo *greedy* si rivelerà più preciso in quanto ad ogni *step* si limita a proporre uno ed un solo arco, ossia l'arco migliore in termini del decremento del *grado di controversia* che consentirebbe se fosse accettato; inoltre, ad ogni *step* dell'algoritmo la scelta dell'arco migliore viene condotta solo dopo aver aggiunto al grafo tutti gli archi consigliati negli *steps* precedenti.

Al contrario, l'algoritmo *non-greedy* propone in un solo passo i k archi migliori utilizzando come metrica il decremento del *grado di controversia* che ciascuno di essi apporterebbe se fosse aggiunto *individualmente*: poiché viene valutato il loro impatto *individuale* e viene ignorato il fatto che tale impatto potrebbe decrementare rispetto a quanto valutato man mano che essi vengono aggiunti al grafo, quest'algoritmo rappresenta un'approssimazione dell'algoritmo *greedy* e consente, in generale, un decremento minore del *grado di controversia*. D'altra parte l'algoritmo *greedy* richiede di scansionare tutti gli archi considerati k volte (una volta per *step*) mentre l'algoritmo *non-greedy* una volta sola: questo si traduce in un sostanziale vantaggio dal punto di vista dei tempi di esecuzione, ottenibile utlilizzando l'algoritmo *non-greedy*.

Per finire, va sottolineato che, in generale, gli archi che vengono scelti dall'*edge-recommendation system* nella realtà non sempre si materializzano (l'utente potrebbe rigettare il consiglio) e per questo è opportuno considerare come metrica anche la *probabilità di accettazione*. Con il proposito di future estensioni, compresa l'introduzione di tale *probabilità*, il *framework* proposto è implementato in modo da prestarsi perfettamente all'aggiunta di altre metriche per la scelta degli archi.

Indice

Sommario II

1 Introduzione 1

1.1 Organizzazione della tesi 7

2 Teoria alla base del problema ed algoritmi per la risoluzione 8

2.1 Misura del grado di controversia 8

2.2 Definizione formale degli algoritmi per la risoluzione 14

2.3 Calcolo del decremento della controversia associato ad un arco 18

3 Raccolta dati ed implementazione 21

3.1 Raccolta dati 22

3.1.1 Twitter Api 22

3.1.2 Tweepy 24

3.1.3 GetOldTweets 25

3.1.4 Processo di raccolta dati 26

3.2 Implementazione 31

3.2.1 Creazione del *retweet graph* 32

3.2.2 Individuazione delle *echo-chambers* 35

3.2.3 Calcolo dell'*RWC* 37

3.2.4 Implementazione degli algoritmi proposti 39

3.2.5 Strumento per la visualizzazione degli archi consigliati 43

4 Test dell'*edge-recommendation system* in modalità *greedy e non* 46

4.1 Discesa dell'*RWC* 47

4.2 Qualità degli archi proposti 51

4.3 Tempi di esecuzione 56

5 Conclusioni e sviluppi futuri 64

Riferimenti bibliografici 65

Elenco delle figure

3.1 Processo di raccolta dati. 30
3.2 Esempio di file che descrive un *retweet graph*. 30
3.3 *Pipeline* implementativa. 31
3.4 Processo di calcolo del *random-walk controversy score* di un *retweet graph*. 37
3.5 *Output* tipico del *tool* di visualizzazione. 45

4.1 Porzione dell'*output* del *tool* a seguito dell'esecuzione di *greedy* sul *retweet graph #beefban*. 58
4.2 Porzione dell'*output* del *tool* a seguito dell'esecuzione di *non greedy* sul *retweet graph #beefban*. 59
4.3 Porzione dell'*output* del *tool* a seguito dell'esecuzione di *greedy* sul *retweet graph #indiana*. 60
4.4 Porzione dell'*output* del *tool* a seguito dell'esecuzione di *non greedy* sul *retweet graph #indiana*. 61
4.5 Porzione dell'*output* del *tool* a seguito dell'esecuzione di *greedy* sul *retweet graph #russia_march*. 62
4.6 Porzione dell'*output* del *tool* a seguito dell'esecuzione di *non greedy* sul *retweet graph #russia_march*. 63

Capitolo 1

Introduzione

Il fenomeno della *polarizzazione* degli utenti attorno a *topic controversi* che si propongono nei *social media* è ben noto ed il suo studio è già stato affrontato in alcuni articoli scientifici (tra cui [2][1]). Potremmo definire la *polarizzazione* come una situazione che determina la divisione della popolazione in gruppi con punti di vista opposti riguardo ad un certo argomento.

Molto spesso i *social media*, mediante algoritmi di *recommendation*, espongono gli utenti solo a contenuti che si addicono e sono conformi alle loro opinioni e, pertanto, non fanno altro che aggravare il loro stato di polarizzazione. Tutto ciò determina la formazione delle così dette *echo-chambers*, ossia situazioni in cui individui che hanno lo stesso parere su un certo argomento rafforzano l'opinione reciproca ma non vengono esposti ad opinioni opposte alla propria.

Lo scopo di questa tesi è quello di sviluppare un sistema in grado far comunicare, nel modo più efficace possibile, queste *echo-chambers* così da esporre gli individui a punti di vista opposti ai propri e ridurre la *controversia* della discussione. L'ambiente in cui opera il *framework* proposto è il *social network* di *Twitter*, in cui gli argomenti delle discussioni vengono identificati da *hashtags*, i contenuti relativi vengono espressi dagli utenti attraverso i *tweets* e le condivisioni di opinione attraverso i

retweets. Una discussione nell'ambiente di *Twitter* può essere descritta mediante un *endorsement graph*, ossia un grafo i cui nodi sono utenti che hanno espresso almeno un'opinione mediante un *tweet* ed i cui archi rappresentano i *retweets*[1].
Il *framework* acquisisce da *Twitter* i dati necessari per elaborare *endorsement graphs* di *topic controversi*, ne analizza la struttura estraendone le *echo chambers* ed infine implementa un *edge-recommendation system* che ha lo scopo di ridurre il *grado di controversia* creando connessioni (*bridges*) tra utenti che hanno punti di vista opposti (appartengono a *echo chambers* distinte). Infatti la naturale propensione degli individui a dare credito solo a notizie e contenuti che si addicono al proprio parere fa sì che, in assenza di un intervento esterno di *edge-recommendation*, essi rafforzino sempre più la propria convinzione, anche qualora questa fosse sbagliata o acritica.

Il sistema proposto in questo lavoro di tesi si compone di più fasi successive, attraverso le quali raggiunge l'obiettivo preposto; nel seguito vengono descritte sinteticamente tali fasi, fornendo le nozioni teoriche e le assunzioni sulle quali si fonda la loro implementazione.

La prima fase si occupa dell'acquisizione dei dati da *Twitter* e quindi della costruzione dell'*endorsement graph* associato. Il *framework* utilizza la libreria *Python Tweepy* (che accede all'*API* di *Twitter*) per ottenere tutti i *tweets* e *retweets* emessi dagli utenti, riguardanti l'*hashtag* fornito in input, in un certo intervallo di tempo. Al termine della collezione di tali dati, essi vengono *parsati* per la costruzione dell'*endorsement graph* che descrive la discussione. Gli *endorsement graphs*, nel particolare ambiente di *Twitter*, prendono anche il nome di *retweet graphs*. Dato un

[1] Vi è un arco da un nodo u ad un nodo v se e solo se l'utente u ha *retweettato* almeno un *tweet* di v.

hashtag controverso, viene a formarsi naturalmente una discussione a riguardo, nella quale gli utenti esprimono una propria opinione e possono approvare il punto di vista delle così dette *autorità*: nel caso particolare di *Twitter* questa *approvazione* si realizza mediante lo strumento del *retweet*, ossia se l'utente u fa *retweet* di un *tweet* prodotto dall'utente v allora ne approva l'opinione. Ne deriva la formazione di un grafo diretto *G(V,E)* costituito da n nodi (gli utenti che partecipano alla discussione) ed i cui archi (*retweets*) esprimono relazioni di condivisione di punti di vista: proprio questo grafo rappresenta l'*output* di questa fase.

La seconda fase si occupa di rilevare le *echo-chambers* dell'*endorsement graph* in input e di calcolarne il *grado di controversia*. Le *echo-chambers* sono due sottoinsiemi dei nodi del grafo X, Y, ben separati tra loro (vi sono pochi archi che li congiungono) e tali che $X \cup Y = V$ e $X \cap Y = \emptyset$. Tale ripartizione dei nodi può essere ottenuta mediante l'utilizzo di un algoritmo di *graph-partitioning*. Nel lavoro proposto è stato utilizzato l'algoritmo di *Girvan-Newman*: esso è un metodo gerarchico usato per rilevare le comunità in sistemi complessi e la cui esecuzione produce un *dendrogramma*, le cui foglie sono i nodi del grafo. Individuate le *echo-chambers*, questa fase si occupa di quantificare la *controversia* della discussione.
Il sistema sviluppato utilizza una metrica basata sul concetto di *random walk* per misurare il *grado di controversia* associato al *topic* analizzato (attorno al quale si svolge la discussione nel *social network* di *Twitter*). Più precisamente, per misurare il *grado di controversia* della rete, viene utilizzata la funzione *Random-walk controversy score*:

$$RWC(G,X,Y) = (c_x - c_y)^T (r_x - r_y)^2$$

[2]La definizione del *Random-walk controversy score* è tratta dall'articolo [4].

Dove c_x è un vettore di dimensione n (numero di vertici dell'*endorsement graph*) che ha valore *1* nelle coordinate corrispondenti ai vertici di grado alto dell'*echo-chamber* X e *0* altrove; similmente viene definito c_y. Infine r_x è il vettore di *PageRank* personalizzato per un *random walk* che parte dai nodi dell'*echo-chamber* X; similmente viene definito r_y. Valori alti di *RWC(G,X,Y)* indicano che, all'equilibrio del *random walk*, è bassa la probabilità di essere nell'*echo-chamber* opposta a quella di partenza: questo è indice di elevata *controversia*.

La terza fase ha lo scopo di ridurre il *grado di controversia* rilevato nella fase precedente mediante l'esecuzione di un *edge-recommendation system*. In particolare, il sistema implementato permette, utilizzando in alternativa un algoritmo *greedy* o uno *non-greedy*, di proporre k archi (che farebbero da *bridges* tra le due *echo chambers*) in grado di ridurre questo indice; l'*edge-recommendation system* proposto restituisce archi la cui efficacia approssima quella degli archi che sono soluzione del problema di ottimizzazione originario che, come detto precedentemente, ha una complessità di un livello troppo elevato ($O(\binom{n^2}{k})$) per essere risolto in tempi accettabili. Di seguito la definizione del problema di ottimizzazione originario:

$$\begin{aligned} \underset{E_\mathrm{k}}{\text{minimize}} \quad & RWC(G(V, E \cup E_\mathrm{k}), X, Y) \\ \text{subject to} \quad & E_\mathrm{k} \subseteq V \times V \setminus E, |E_\mathrm{k}| = k \end{aligned}$$

Ossia il problema originario consiste nel trovare l'insieme di k archi, considerando *tutti* gli archi non ancora presenti nel grafo, che se si materializzassero *minimizzerebbero* l'*indice di controversia*. L'euristica proposta in questo lavoro permette di restringere il dominio degli archi considerati, consentendo di ottenere risultati paragonabili con quelli ottenibili mediante la soluzione del problema di ottimizzazione

appena descritto e con il vantaggio di avere una complessità computazionale di molto inferiore; essa, forniti in *input* i valori di k_1 e k_2 (interi positivi):

1. considera i k_1 vertici con *in-degree*[3] più alto dell'*echo-chamber* X e i k_2 vertici con *in-degree* più alto dell'*echo-chamber* Y;

2. costruisce il dominio degli archi considerati come l'insieme di tutti i possibili archi diretti, non presenti ancora nel grafo, che abbiano come estremi un vertice dell'insieme K_1 e uno dell'insieme K_2;

Gli algoritmi *greedy* e *non-greedy* operano entrambi a partire dal dominio così definito. Hanno lo scopo di proporre i k archi più promettenti del dominio in termini del decremento del grado di controversia *RWC(G,X,Y)* che consentirebbero qualora si materializzassero nel grafo; la differenza sta nella modalità di selezione. In particolare:

1. l'algoritmo *greedy* sceglie *avidamente* ognuno dei k archi: esso impiega k passi, in ognuno dei quali sceglie l'arco più appetibile del dominio (ossia l'arco del dominio non ancora presente nel grafo associato al δRWC maggiore) e lo aggiunge al grafo.

2. l'algoritmo *non-greedy* ordina una sola volta tutti gli archi del dominio secondo il δRWC che ognuno consente e sceglie in un solo passo i k archi migliori. Ne deriva una maggiore efficienza nei tempi di esecuzione ma una minore precisione nella scelta degli archi da proporre.

In pratica, per ridurre la *controversia*, si propone ad un certo insieme di utenti il contenuto (i.e. *tweets*) di utenti che hanno posizioni sull'argomento opposte rispetto alle proprie, sperando che la maggior parte di loro possa farne *endorsement* mediante

[3]Il numero di archi diretti del grafo che hanno come nodo di destinazione un nodo x è detto *in-degree* di x.

lo strumento del *retweet*: ciò provocherebbe la formazione di nuovi archi tra le due comunità (*echo chambers*) con l'effetto di ridurre il *grado di controversia* dell'intero *retweet graph*.

Il *framework* implementato è stato testato su tre *retweet graphs* di tre *topics* (ossia *hashtags*) controversi di *Twitter*: *#beefban,#russia_march,#indiana*. I test sono stati prodotti seguendo l'approccio del *paper*[4], che ha lo scopo di mostrare il livello di *controversia* del grafo in funzione della quantità di archi al momento aggiunti. Tuttavia l'obiettivo dei test, in questa tesi, è in primo luogo quello di fornire un confronto tra l'efficacia dell'*edge-recommendation system* basato sull'algoritmo *greedy* e l'efficacia dell'*edge-recommendation system* basato sull'algoritmo *non-greedy*.
Per *efficacia* si intende, fissato come obiettivo il decremento del grado di controversia di una quantità ΔRWC, la quantità minima di archi, consigliati dal sistema, che gli utenti devono accettare per poter raggiungerlo.
In secondo luogo i test hanno lo scopo di fornire un raffronto dei tempi di esecuzione dei due algoritmi al variare dell'entità del grafo in input, entità espressa in termini del numero di nodi n e del numero di archi e.

Per terminare, il *framework* offre un *tool* per la visualizzazione degli archi proposti, evidenziando le caratteristiche dei nodi coinvolti tra cui l'*in degree* (ossia il grado in ingresso); inoltre è utilizzato un algoritmo di *coloring*[4] per classificare tali nodi colorandoli in modo diverso, in funzione dell'*echo-chamber* a cui appartengono.
Di seguito è illustrata brevemente l'organizzazione della tesi.

[4]Con *coloring* si intende una colorazione esatta dei vertici, cioè un'etichettatura dei vertici del grafo con colori tali che nessuna coppia di vertici che condividono lo stesso arco abbiano lo stesso colore.

1.1 Organizzazione della tesi

L'esposizione del lavoro di tesi ha l'obiettivo di fornire *dettagli* riguardanti:

- *la teoria che è alla base del problema affrontato;*
- *la raccolta dei dati e l'implementazione del framework;*
- *le modalità in cui sono stati effettuati i test ed i risultati ottenuti;*
- *sviluppi futuri.*

Gli approfondimenti teorici verranno illustrati nel capitolo *Teoria alla base del problema ed algoritmi per la risoluzione*. Il capitolo *Raccolta dati ed implementazione* si occuperà di fornire dettagli sulle tecnologie utilizzate per la raccolta dei dati e sull'implementazione del *framework*. I risultati sperimentali ottenuti dai test condotti (nelle modalità *greedy e non*) e le osservazioni ad essi riguardanti sono trattati nel capitolo *Test dell'edge-recommendation system in modalità greedy e non*. Il capitolo *Conclusioni e sviluppi futuri* trarrà le conclusioni a partire dai risultati dei test ed approfondirà le sfide ed i propositi di miglioramento del sistema implementato.

Capitolo 2

Teoria alla base del problema ed algoritmi per la risoluzione

2.1 Misura del grado di controversia

Prima di dare una definizione formale del *random-walk controversy score*, elenchiamo ed illustriamo i passi necessari per calcolarlo.

1. Fissato il *topic t* per il quale si vuole quantificare il grado di controversia, è possibile descrivere la discussione mediante l'*endorsement graph G(V,E)*. Nell'ambiente di *Twitter*, il *topic t* è identificato da un *hashtag* (e.g.*#hashtag*) ed i nodi del grafo rappresentano gli utenti che hanno preso parte alla discussione utilizzando almeno una volta tale *hashtag* nei loro *tweets*; gli archi del grafo identificano i *retweets* tra gli utenti, che esprimono relazioni di condivisione di opinione riguardo al *topic*.

2. Ipotizzando che il *topic t* sia controverso, è possibile partizionare i nodi del grafo *G(V,E)* in due insiemi X, Y ben separati tra loro (i.e. vi sono pochi archi che li interconnettono). Tali insiemi quindi soddisfano le seguenti proprietà:

(a) $X \cup Y = V$;

(b) $X \cap Y = \emptyset$.

Gli insiemi X ed Y rappresentano i due lati della controversia (i.e. le *echo-chambers*).

Per identificare le *echo-chambers*, nell'implementazione proposta è stato utilizzato l'algoritmo di *graph-partitioning* di *Girvan-Newman*[5]. Tale algoritmo agisce rimuovendo progressivamente archi dal grafo originario: l'esecuzione viene arrestata quando la rimozione degli archi ha portato ad individuare due comunità distinte che non comunicano (i.e. non sono collegate da nessun arco).

La metrica utilizzata da *Girvan-Newman* per identificare l'arco da rimuovere ad ogni passo è la così detta *edge-betweenness centrality*: dato un arco e, essa è definita come *il numero di cammini di costo minimo tra coppie di nodi del grafo che passano attraverso l'arco e*. Nel caso in cui vi sia più di un percorso di costo minimo tra una coppia di nodi, a ciascun percorso viene assegnato uguale peso in modo tale che il peso totale di tutti i percorsi sia uguale all'unità. Di seguito la formula che definisce questa metrica di centralità:

$$b(e) = \sum_{s \neq t} \frac{\sigma_{\mathrm{st}}(e)}{\sigma_{\mathrm{st}}} \tag{2.1}$$

Dove σ_{st} è il numero totale di percorsi di costo minimo dal nodo s al nodo t e $\sigma_{st}(e)$ è il numero di tali percorsi che passano attraverso l'arco e.

L'intuizione è: se la struttura del grafo è caratterizzata da due comunità di nodi connesse tra loro da pochissimi archi, allora tutti i percorsi tra queste due comunità dovranno passare attraverso questi archi. Ne consegue che quest'ultimi saranno caratterizzati da un'alta *betweenness centrality*. Sfruttando

la peculiarità di tali archi, l'algoritmo di *Girvan-Newman* si rivela un ottimo metodo per rilevare le *echo-chambers*.

3. A questo punto è possibile procedere con la definizione del *random-walk controversy score*. L'*RWC* è definito come *la differenza della probabilità che un random walk che parte da una echo-chamber all'equilibrio vi permanga e la probabilità che invece tale random walk all'equilibrio finisca nell'echo-chamber opposta*. Tale misura viene calcolata mediante l'utilizzo di due esecuzioni dell'algoritmo di *PageRank* personalizzato, le quali non sono altro che due *random walks* particolari.

   PageRank è un algoritmo di analisi che assegna un peso numerico a ciascun nodo di un grafo diretto, con lo scopo di quantificare la sua importanza relativa. Le applicazioni più frequenti di *PageRank* riguardano l'ambito del *World Wide Web*, in cui i grafi hanno come nodi le pagine *web* e come archi i collegamenti ipertestuali. Ciò non toglie che *PageRank* sia uno strumento molto potente anche nell'ambito delle reti sociali, poiché riesce a quantificare l'importanza di utente nell'ambito di una discussione: tale importanza misura il suo grado di popolarità e di rilevanza.

   Per il calcolo dell'*RWC* vengono utilizzate due esecuzioni distinte dell'algoritmo di *PageRank*, indicate con $page_x$ *e* $page_y$, ognuna delle quali opera sul *retweet graph* corrispondente al *topic* t in input ma inizia il suo *random-walk* partendo, rispettivamente, da uno dei nodi della *comunità* X e da uno dei nodi della *comunità* Y (*comunità* = *echo-chamber*). Inoltre, $page_x$ *e* $page_y$ ad ogni passo possono decidere di continuare il *random-walk* (potendo scegliere con uguale probabilità uno degli archi in uscita dal nodo in cui si trovano attualmente) o di ricominciare il proprio cammino (*restart*), tornando, rispettivamente, in uno dei nodi della *comunità* X ed in uno dei nodi della *comunità* Y: la seconda scelta viene compiuta con una probabilità detta di *restart* e,

chiaramente, la prima con una probabilità che ne è il complementare (i.e. la somma delle probabilità deve restituire *1*).
Scendendo più nel dettaglio, siano:

- P la *matrice delle probabilità di transizione per colonna*[1] associata al *retweet graph* considerato;
- X^* *e* Y^* rispettivamente gli insiemi dei k_1 *e* k_2 nodi con *in-degree* più alto delle due comunità X *e* Y. Inoltre sia c_x un vettore di dimensione n avente valore *1* nelle coordinate corrispondenti ai nodi dell'insieme X^* e *0* altrove; similmente viene definito c_y;
- r_x il vettore di *PageRank* personalizzato per il *random walk* che parte dalla comunità X. Sia inoltre $(1-\alpha)$ la probabilità di *restart* di tale *random walk* (e dunque α è la probabilità di continuare) e sia $e_x = Uniform(X)$ il suo vettore di *restart*: ossia il *random walk*, ad ogni passo, decide di ricominciare il proprio cammino con probabilità $(1-\alpha)$ e tra tutti i nodi della *comunità* X, con eguale probabilità, sceglie il nodo da cui ricominciare.
  Simili considerazioni valgono per r_y.

Bisogna ora risolvere il problema dei vertici *dangling*, ossia i vertici del grafo che non hanno archi in uscita. Se un *random walk* dovesse casualmente finire in uno di questi nodi esso potrebbe non uscirne, compromettendo l'esecuzione di *PageRank*. Per evitare tale situazione, vengono utilizzate non una ma due *matrici delle probabilità di transizione per colonna* P_x *e* P_y, usate rispettivamente dal *random walk* che inizia dalla comunità X e dal *random walk* che

[1] Se il *retweet graph* in considerazione ha N nodi, tale matrice ha dimensione $N \times N$ ed ogni suo elemento *P[i][j]* è la probabilità di passare dal nodo j al nodo i in un solo passo, sapendo di essere attualmente nel nodo j.

inizia dalla comunità *Y*. Se il grafo non contiene vertici *dangling*, si ha banalmente $P_x = P_y = P$; se al contrario li contiene, le matrici P_x *e* P_y sono definite in modo tale che le probabilità di transizione dai vertici *dangling* sono uguali, rispettivamente, ai vettori di *restart* e_x *ed* e_y.
Il *PageRank* personalizzato per i due *random walks* che, rispettivamente, iniziano nella *comunità X e Y* e dato da:

$$r_x = \alpha P_x r_x + (1 - \alpha) e_x \tag{2.2}$$

$$r_y = \alpha P_y r_y + (1 - \alpha) e_y \tag{2.3}$$

Per il calcolo dei vettori di *PageRank* personalizzati (r_x *e* r_y), quindi, bisogna imporre una situazione di *stazionarietà* dei rispettivi *random walks*.
Possiamo finalmente definire il *random-walk controversy score*:

$$RWC(G, X, Y) = (c_x - c_y)^{\mathrm{T}}(r_x - r_y) \tag{2.4}$$

Sostituendo le equazioni (2.2) e (2.3) nell'espressione (2.4) si ottiene:

$$RWC(G, X, Y) = (1 - \alpha)(c_x - c_y)^{\mathrm{T}}(M_x^{-1} e_x - M_y^{-1} e_y) \tag{2.5}$$

Dove $M_x = (I - \alpha P_x)$ e $M_y = (I - \alpha P_y)$. Quest'ultima è la formula utilizzata nell'implementazione del sistema proposto. Ad ogni modo, riferendosi all'equazione (2.4) è possibile fare le seguenti osservazioni:

(a) Innanzitutto occorre precisare che r_x è una *distribuzione stazionaria di probabilità* del *random walk* associato all'esecuzione di *PageRank* $page_x$: ognuno dei suoi elementi $r_x[i]$ ($\forall i = 0,\ldots,n-1$) rappresenta la probabilità

che il *random walk* si trovi nel nodo i all'equilibrio. Simili considerazioni valgono per r_y;

(b) Qualora le due comunità (*echo-chambers*) X *e* Y fossero molto divise tra loro (i.e. pochissimi archi diretti che le connettono), ci si aspetterebbe un r_x con valori pressoché nulli nelle coordinate corrispondenti ai nodi di Y e non nulli altrove e, viceversa, parlando di r_y[2]. In tal caso l'espressione $(c_x - c_y)^{\mathrm{T}}(r_x - r_y)$ assume il valore massimo, indice di un'elevata *controversia* nella rete.

(c) Qualora le due comunità fossero invece sufficientemente connesse tra loro, ci si aspetterebbe un vettore r_x con valori abbastanza uniformi su tutte le sue coordinate; la stessa considerazione varrebbe per r_y. In tal caso il grafo non presenterebbe un'elevata controversia (*RWC(G,X,Y)* assume valori bassi), e questo risultato sarebbe confermato dal fatto che le due comunità sono sufficientemente esposte l'una a l'altra.

Per terminare, si può affermare che l'indice *RWC(G,X,Y)* riesce a descrivere perfettamente il grado di controversia della discussione in atto, in funzione del livello di esposizione reciproca delle due comunità (ossia i due punti di vista opposti): qualora tale livello fosse molto basso, l'indice di controversia sarebbe molto elevato e, pertanto, la scelta più efficace per attenuarlo sarebbe quella di cercare di esporre gli utenti a visioni opposte alle proprie.
Nel prossimo paragrafo saranno illustrati gli algoritmi utilizzati nel sistema proposto, i quali hanno come obiettivo proprio quello di esporre reciprocamente, *nel modo più efficace possibile*, i due lati della controversia.

[2]Ricordare che, essendo *distribuzioni di probabilità*, vale: $\sum_{i=0}^{n-1} r_x[i] = 1$ e $\sum_{j=0}^{n-1} r_y[j] = 1$.

2.2 Definizione formale degli algoritmi per la risoluzione

Il problema che si vorrebbe risolvere è il seguente:

Dato un endorsement graph G(V,E) che descrive una discussione riguardo ad un certo topic controverso t, trovare l'insieme di k archi diretti non ancora presenti nel grafo che, qualora si materializzassero, minimizzerebbero il suo RWC(G,X,Y).

Ovvero, formalmente:

$$\begin{aligned} \underset{E_\mathrm{k}}{\text{minimize}} \quad & RWC(G(V, E \cup E_\mathrm{k}), X, Y) \\ \text{subject to} \quad & E_\mathrm{k} \subseteq V \times V \setminus E, |E_\mathrm{k}| = k \end{aligned}$$

Risolvere tale problema, così come si presenta, richiederebbe di considerare tutti le possibili combinazioni degli archi ancora non presenti nel grafo presi a gruppi di k: queste combinazioni sono pari a $O(\binom{n^2}{k})$, dove con n indichiamo il numero di nodi del grafo.
Come è facile immaginare, un algoritmo che considera tutte queste combinazioni di archi è molto inefficiente dal punto di vista computazionale, soprattutto alla luce del fatto che gli *endorsement graphs* delle reti sociali come quella di *Twitter* sono costituiti da alcune migliaia di nodi e migliaia di archi.
Si propone, pertanto, di restringere il dominio degli archi candidati: seguendo l'approccio dell'articolo [4] è possibile considerare solo gli archi tra vertici con *in-degree* alto di ciascuna *echo-chamber*. La scelta di questa euristica, come osservano gli autori dell'articolo [4], è giustificata dal fatto che la struttura degli *endorsement graph* spesso mette in luce la presenza di un piccolo numero di nodi *leader* (i.e. utenti

popolari) ed un gran numero di nodi *non leader* (i.e. utenti seguaci): i nodi *leader* ricevono molti archi in ingresso (ossia hanno alto *in-degree*) in quanto i loro *tweet* vengono *retweettati* da molti nodi *non leader* (ossia ricevono molta *approvazione* dai loro seguaci, i quali si fidano della loro opinione). Intuitivamente, qualora si riuscisse a convincere i nodi *leader* ad approvare contenuti che esprimono opinioni sul *topic controverso* opposte alla propria, i nodi *non leader* (i seguaci) sarebbero spinti a fare altrettanto. Questa osservazione suggerisce che gli archi tra vertici con *in-degree* alto di ciascuna *echo-chamber* sono buoni candidati per ottenere un abbassamento del *random-walk controversy score*[3].

Mediante questo ridimensionamento del dominio degli archi candidati, ci si propone di ridurre l'*RWC* esponendo alcuni nodi *leader* di ciascuna *echo-chamber* ai contenuti di alcuni nodi *leader* dell'*echo-chamber* opposta.

Ora, con l'obiettivo di effettuare la scelta dei k archi migliori appartenenti al dominio così definito, vengono proposti due algoritmi: un *algoritmo non greedy* (indicato di seguito come *Algorithm 1*) ed un *algoritmo greedy*.

L'algoritmo *non greedy* ha un tempo di esecuzione pari a $O(k_1 \cdot k_2)$, che costituisce un ottimo *speedup* rispetto all'algoritmo *brute-force*, che considera invece tutte le combinazioni di archi. *Algorithm 1* si limita a scegliere dal dominio degli archi, ridimensionato come appena detto, i k archi migliori in termini del decremento dell'*RWC* che ciascuno consente se aggiunto *individualmente*. Tuttavia il *decremento dell'RWC* che un qualsiasi arco $''e''$ consente individualmente (δRWC_e) si rivelerebbe tale solo se tale arco fosse aggiunto per primo al grafo; più precisamente $\forall E' \supseteq E, \forall e \in V \times V \setminus E'$ vale l'espressione (2.6).

$$RWC(G(V,E),X,Y) - RWC(G(V,E \cup \{e\}),X,Y) \geq RWC(G(V,E'),X,Y) - RWC(G(V,E' \cup \{e\}),X,Y) \tag{2.6}$$

[3]Si rimanda alla lettura del *Teorema 1.* dell'articolo [4].

Ovvero la somma dei δRWC dei k archi scelti dall'algoritmo *non greedy* non corrisponde al decremento *reale* dell'RWC che si osserverebbe se tali archi apparissero nel grafo, ma ne è un *upper-bound*. Questo significa che l'algoritmo *non greedy* è poco preciso nella scelta e potrebbe pertanto disattendere le aspettative di decremento della *controversia*.
L'alternativa è l'utilizzo di una versione *greedy* di tale algoritmo. L'algoritmo *greedy* non fa altro che effettuare la scelta dei k archi non in un solo *step* ma in k *step*. In ognuno dei k passi sceglie uno ed un solo arco, ossia l'arco migliore, tra quelli ancora disponibili, in termini del δRWC che consentirebbe se fosse aggiunto al grafo: l'arco scelto, infine, viene aggiunto al grafo con l'obiettivo di consentire una scelta più precisa dei restanti archi.
Si osserva che, per la versione *greedy*, è vero che la somma dei δRWC dei k archi scelti corrisponde al decremento *reale* dell'RWC che si osserverebbe se tali archi apparissero nel grafo. Nel capitolo relativo ai *test* sarà possibile osservare quanto la versione *greedy* riesca ad individuare archi migliori, in termini del decremento dell'RWC, rispetto a quelli individuati dalla versione *non greedy*.
Lo svantaggio della versione *greedy* riguarda il tempo di esecuzione. Il fatto che tale algoritmo sia sintetizzabile come un algoritmo *non greedy* eseguito k volte implica che il suo tempo di esecuzione sia pari a $O(k \cdot k_1 \cdot k_2)$, ovvero k volte il tempo di esecuzione della versione *non greedy*. Ad ogni modo, la scelta dell'algoritmo di *edge recommendation* da utilizzare dovrebbe sempre essere dettata da un giusto compromesso tra i risultati che consente di ottenere e l'efficienza nei tempi di esecuzione.

Nel prossimo paragrafo sarà illustrata una tecnica efficiente per calcolare il δRWC associato a ciascun arco considerato dagli algoritmi appena descritti.

Algorithm 1 Algoritmo *non greedy* per la scelta dei k archi

Require: Il grafo G e le comunità X, Y; il numero di archi da proporre k; i k_1 *e* k_2 vertici con *in-degree* più alto in X *e* Y, rispettivamente

Ensure: La lista dei k archi migliori, in termini del decremento dell'RWC che consentono se aggiunti individualmente

Initialization : Output ← lista vuota;
for i = *1:*k_1 **do**
  nodo $u = X[i]$;
  for j = *1:*k_2 **do**
    nodo $v = Y[j]$;
    Calcola il decremento dell'RWC $\delta RWC_{u \to v}$ che si otterrebbe qualora l'arco *(u,v)* venisse aggiunto al grafo;
    Aggiungi $\delta RWC_{u \to v}$ alla lista *Output*;
    Calcola il decremento dell'RWC $\delta RWC_{v \to u}$ che si otterrebbe qualora l'arco *(v,u)* venisse aggiunto al grafo;
    Aggiungi $\delta RWC_{v \to u}$ alla lista *Output*;
  end for
end for
Lista *Output* ordinata ← Ordina la lista *Output* secondo i δRWC (cambiati di segno) in ordine decrescente;
return I migliori k dalla lista *Output* ordinata;

2.3 Calcolo del decremento della controversia associato ad un arco

Per ogni arco $e \in V \times V \setminus E$ appartenente al dominio dei $2 \times k_1 \times k_2$ archi diretti considerati dagli algoritmi *non greedy e greedy*, si rende necessario calcolare il δRWC_e che esso consentirebbe qualora venisse aggiunto al grafo. Per effettuare questo calcolo verrebbe in mente di aggiungere l'arco *"e"* al grafo, calcolare il nuovo $RWC_{\{e\}}$, per poi ricavare:

$$\delta RWC_e = RWC \text{ - } RWC_{\{e\}} \tag{2.7}$$

È bene ricordare che il calcolo dell'RWC è abbastanza oneroso e pertanto è meglio evitarlo quando possibile: se si utilizzasse la modalità (2.7), gli algoritmi di scelta dei k archi dovrebbero calcolare l'RWC un numero di volte pari a $2 \times k_1 \times k_2$, cagionando un degrado prestazionale non trascurabile.
Tuttavia, visto che siamo interessati solamente all'entità del decremento dell'RWC a seguito dell'aggiunta di un arco, possiamo utilizzare un'altra modalità di calcolo più efficiente. In particolare, considerando la *matrice delle probabilità di transizione per colonna* P, dopo l'aggiunta del generico arco diretto *(i,j)* solo una sua colonna ne è affetta: la colonna che corrisponde al vertice di origine *"i"* dell'arco *(i,j)*.
Di seguito è evidenziata la colonna *i-esima* prima e dopo l'aggiunta dell'arco diretto *(i,j)*:

$$P^T = \begin{bmatrix} \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \frac{1}{q} & \frac{1}{q} & \cdots & \frac{1}{q} & 0 & 0 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \end{bmatrix}$$

Dove q è l'*out-degree* del nodo *i*. A seguito dell'aggiunta dell'arco diretto *(i,j)* la colonna *i-esima* diventa:

$$P'^T = \begin{bmatrix} \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \frac{1}{q+1} & \frac{1}{q+1} & \cdots & \frac{1}{q+1} & \frac{1}{q+1} & 0 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \end{bmatrix}$$

L'elemento in posizione *j-esima* della colonna *i-esima* della nuova *matrice di transizione P'* passa da un valore pari a *0* ad un valore pari a $\frac{1}{q+1}$.
Sia ora u^T l'*i-esimo* vettore della base standard di $\mathbb{R}^n$ (dove n è il numero di nodi del grafo); similmente, sia v^T il *j-esimo* vettore della base standard di $\mathbb{R}^n$.
Definiamo infine il vettore z^T come segue:

1. Se il vertice i non è un vertice *dangling*, $z^T = \frac{1}{q+1}[\frac{1}{q}, \frac{1}{q}, ..., \frac{1}{q}, -1, ...0,0,0]$, con *-1* nella posizione corrispondente al vertice di arrivo j;

2. Se il vertice i è un vertice *dangling*, $z^T = e_x - v$ oppure $z^T = e_y - v$, rispettivamente se si considera la matrice P_x (*random walk* che parte dall'*echo-chamber X*) o la matrice P_y (*random walk* che parte dall'*echo-chamber Y*).

Si dimostra che la *matrice delle probabilità di transizione* aggiornata è data da:

$$P' = P - z \otimes u^T \tag{2.8}$$

Dove il simbolo $\otimes$ indica che il prodotto tra i due vettori non è scalare ma *esterno*, il cui risultato è una matrice.
Ricordando dalla definizione dell'*RWC* (2.5) le formule che descrivono le matrici M_x

ed M_y e sostituendovi, rispettivamente, P'_x e P'_y (seguendo la formula (2.8)) si ottiene:

$$M'_x = M_x + \alpha z_x \otimes u^T \tag{2.9}$$

$$M'_y = M_y + \alpha z_y \otimes u^T \tag{2.10}$$

Nell'implementazione proposta, l'inversa delle matrici M'_x e M'_y (inversa che occorre per il calcolo dell'*RWC*) è calcolata usando la formula di *Sherman-Morrison*[3], vista la sua efficienza.
Ora, per l'equazione (2.5), l'*RWC* del grafo a seguito dell'aggiunta del nuovo arco diretto può essere scritto come:

$$RWC' = (1-\alpha)(c_x - c_y)^{\mathrm{T}}(M_x'^{-1}e_x - M_y'^{-1}e_y) \tag{2.11}$$

E, finalmente, il δRWC è dato da:

$$\begin{aligned}\delta RWC = RWC' - RWC = (1-\alpha)(c_x - c_y)^{\mathrm{T}}(-(\frac{\alpha M_x^{-1} z_x \otimes u^T M_x^{-1}}{1+\alpha u^T M_x^{-1} z_x})e_x \\ +(\frac{\alpha M_y^{-1} z_y \otimes u^T M_y^{-1}}{1+\alpha u^T M_y^{-1} z_y})e_y)\end{aligned} \tag{2.12}$$

Quest'equazione, sebbene apparentemente ingombrante, permette di calcolare in modo molto efficiente il δRWC di ogni arco considerato, evitando di dover calcolare ogni volta il nuovo *RWC* (operazione molto costosa) per poi prendere la differenza rispetto all'*RWC* precedente.

Nel prossimo capitolo saranno presentate le tecnologie utilizzate per realizzare il sistema proposto e verranno illustrati i dettagli implementativi.

Capitolo 3

Raccolta dati ed implementazione

Questo capitolo si occuperà, prima di tutto, di fornire dettagli sulle tecnologie e sulle modalità di raccolta dei dati da *Twitter*, necessari alla costruzione di *retweet graphs* associati ad *hashtags* in *input*.
Infine verrà trattata puntualmente l'implementazione degli algoritmi, definiti rigorosamente nel capitolo precedente, e di tutte quelle tecniche che hanno permesso di raggiungere l'obiettivo preposto, ovvero l'implementazione di un *framework* che permetta di:

1. costruire ed analizzare *retweet graphs* associati ad *hashtags di Twitter* in *input*;

2. rilevare le *echo-chambers* che caratterizzano la discussione;

3. eseguire un algoritmo di *k-edge recommendation*, in modalità *greedy* o meno;

4. fornire strumenti per l'analisi degli archi consigliati e per la visualizzazione dei nodi coinvolti (i.e. i nodi estremi degli archi consigliati).

3.1 Raccolta dati

La raccolta dei dati è una fase indispensabile, una condizione *sine qua non*, senza la quale non è pensabile raggiungere alcun obiettivo tra quelli prefissati.
Poiché il *software* proposto si occupa di *endorsement graphs* del *social network* di *Twitter*, per la raccolta dei dati si è reso necessario l'utilizzo della *Twitter Api*. Inoltre, visto che il linguaggio utilizzato per l'implementazione è *Python*, sono state sfruttate le funzionalità della libreria *Tweepy*, una via di accesso alla *Twitter Api* di facile utilizzo.
Nel seguito saranno forniti dettagli sugli strumenti di *Twitter Api e Tweepy* e su altre tecniche che hanno permesso di *bypassare* importanti limitazioni temporali delle *Api di Twitter*.

3.1.1 Twitter Api

Twitter mette a disposizione degli sviluppatori delle *Api*, utili per l'acquisizione dei dati pubblicati dagli utenti. Per rendere possibile il loro utilizzo bisogna innanzitutto creare un *account Twitter* e poi effettuare l'iscrizione al *reparto sviluppatori di Twitter*. Questa procedura è molto rigida e qualora non fosse seguita in modo rigoroso non sarebbe possibile utilizzare le *Api*.
Una volta effettuata l'iscrizione al *reparto sviluppatori di Twitter*, è finalmente possibile procedere con la raccolta dei dati pubblicati dagli utenti, ma non prima di aver ottenuto le credenziali di accesso. Le credenziali vengono rilasciate a seguito della creazione di una *Twitter App*, che rappresenta un *progetto Twitter* dello sviluppatore: esse permetteranno di autenticarsi presso un *server di Twitter*, con il quale sarà possibile interagire *via streaming* mediante le *Twitter Api* per ottenere i dati richiesti. Le credenziali si dividono in *Token* e *Consumer*, i quali hanno le seguenti caratteristiche e funzioni:

- il *Token* permette l'accesso ai servizi che offre *Twitter*. Non è sufficiente per consentire lo *streaming* dei dati dal *server*. È costituito da:
  - *Access Token*;
  - *Access Secret*.
- il *Consumer* consente lo *streaming* dei dati di *Twitter* dal *server*. È costituito da:
  - *Consumer Key*;
  - *Consumer Secret*.

Nell'ambito della stessa *Twitter App*, queste chiavi possono essere rigenerate a piacimento, anche con l'obiettivo di evitare problematiche relavite alla sicurezza. Qualunque sia il linguaggio di programmazione utilizzato per lo sviluppo del *software* (nel caso in esame *Python*), per utilizzare le *Api* da codice è sempre necessario prima autenticarsi, fornendo tutti e quattro i codici appena elencati: le interfacce d'accesso alle *Api* di *Twitter* tuttavia dipendono dal linguaggio di programmazione e, nel nostro caso, sono realizzate mediante la libreria di *Python Tweepy*, della quale parleremo nel seguito della trattazione.
Ad ogni modo, bisogna sottolineare che lo *streaming* dei dati viene limitato da *Twitter* per evitare che gli sviluppatori utilizzino in modo sconveniente i dati pubblicati dagli utenti: uno sviluppatore, pur essendo dotato di tutte le credenziali necessarie, non può effettuare più di 100 richieste ogni 15 minuti. Durante il tempo di pausa, che viene fatto scattare in corrispondenza del superamento della soglia di richieste, lo sviluppatore può decidere di aspettare che esso si esaurisca o, al contrario, può decidere deliberatamente di violarlo ed effettuare una nuova richiesta: in tal caso le sue credenziale verrebbero bloccate e non gli sarebbe permesso di comunicare con il *server* mediante le *Api* per circa un'ora.

Un'altra limitazione che impone l'*Api* ufficiale di *Twitter* riguarda l'impossibilità di ottenere *tweets* più vecchi di una settimana: questa limitazione è molto forte ed ha costituito, nel processo di sviluppo del *framework* proposto, un problema molto ingente, la cui risoluzione è dovuta alla libreria *GetOldTweets di Python*, della quale parleremo presto.
Per terminare, i dati che lo sviluppatore richiede al *server* mediante la *Twitter Api* vengono restituiti in un file *JSON*: esso conterrà tutti i *metadati* necessari, i quali dipendono dal criterio della *query*, come ad esempio il testo del *tweet*, gli *hashtags*, lo *username* dell'utente che l'ha pubblicato e gli utenti che l'hanno *retweettato*.

3.1.2 Tweepy

Tweepy è una libreria di *Python* che permette di accedere agevolmente alle *Api* di *Twitter*. Gestisce l'autenticazione dello sviluppatore presso il server di *streaming* utilizzando i seguenti metodi:

1. *tweepy.OAuthHandler(CONSUMER_KEY, CONSUMER_SECRET)*:
   una volta forniti *Consumer Key e Consumer Secret* validi, restituisce un codice di autenticazione *auth*;

2. *auth.set_access_token(ACCESS_TOKEN, ACCESS_TOKEN_SECRET)*:
   permette di impostare il codice *auth* con gli *Access Token e Access Token Secret* (validi);

3. *tweepy.API(auth)*: restituisce, in caso di corretta autenticazione, un oggetto *Api* attraverso il quale può finalmente avvenire il processo di *streaming* dal *server*.

Inoltre *Tweepy* permette di gestire vari tipi di errore tra cui *RateLimitError*, che insorge quando viene superata la soglia di traffico di 100 richieste ogni 15 minuti.

3.1.3 GetOldTweets

Come precedentemente detto, l'*Api* ufficiale di *Twitter* rende impossibile, con un semplice *account* gratuito, l'acquisizione di *tweets* più vecchi di una settimana. Questa limitazione, nel caso del sistema proposto, è intollerabile, visto che, per costruire *retweet graphs* di dimensioni sufficienti a condurre un'analisi significativa, bisogna utilizzare un intervallo di osservazione abbastanza ampio. Superare tale limitazione continuando ad utilizzare l'*Api* ufficiale vorrebbe dire pagare per ottenere un *account Enterprise*, cosa che non siamo disposti a fare.
La libreria *GetOldTweets* permette di *bypassare* l'*Api* ufficiale e di ottenere *tweets* più vecchi di una settimana semplicemente sfruttando la funzione *scroll* della pagina di *Twitter*: facendo *scroll* verso il fondo pagina è possibile ottenere, tramite chiamate successive ad un *provider JSON*, *tweets* (relativi all'*hashtag* che si sta cercando) via via più vecchi, evitando di incorrere a limitazioni temporali. Tale libreria mette a disposizione un gran numero di criteri di ricerca, utilizzati poi come parametri dell'indirizzo *http*. Nel caso in esame sono stati utilizzati i seguenti parametri:

- *Since*: una data limite inferiore per limitare la ricerca;
- *Until*: una data limite superiore per limitare la ricerca;
- *QuerySearch*: il testo di *query* desiderato. Nel caso in esame, come *query* viene sempre specificato un *hashtag*, il quale identifica una discussione, e vengono considerati tutti e soli i *tweets* creati nell'intervallo temporale specificato e che recano tale *hashtag*.

Una volta costruito un oggetto *tweetCriteria*, specificando le informazioni sopra elencate, esso viene passato come parametro al metodo *getTweets* della classe *TweetManager*, il quale si occupa di recuperare tutti i *tweets* che soddisfano i criteri di ricerca. In particolare questo metodo, una volta costruita la *url* contenente tutti i

parametri di ricerca specificati, acquisisce la pagina *web* contenente tutti i *tweets* che soddisfano i criteri e converte il risultato in un formato *JSON*. Le informazioni che, nel caso specifico dell'implementazione proposta, vengono estratte dai *tweets* risultanti sono due:

- *ID* del *tweet*;
- *Username* dell'autore del *tweet*.

Gli *ID* dei *tweets* verranno utilizzati per recuperare tutti i *retweets* associati, i quali sono indispensabili per costruire il *retweet graph*.

3.1.4 Processo di raccolta dati

Descritte le caratteristiche e le peculiarità degli strumenti utilizzati per effettuare la raccolta dei dati, ora occorre analizzare il processo che permette di acquisirli e di renderli persistenti. Con tale obiettivo è stata implementata la classe *TwittersRetweets*. Essa fornisce metodi per:

1. Specificare i parametri di ricerca (i.e. *Since, Until, QuerySearch*);
2. Recuperare dal *social network di Twitter* tutti i *tweets* che soddisfano i parametri di ricerca specificati al punto *1.*, insieme agli utenti che li hanno prodotti;
3. Recuperare tutti i *retweets* che sono stati prodotti verso i *tweets* recuperati al punto *2.*;
4. Organizzare i dati ottenuti in un *file* e renderli persistenti.

Un oggetto della classe *TwittersRetweets* ha pertanto i seguenti attributi:

- *since*, ossia la data di inizio ricerca;

- *until*, ossia la data di fine ricerca;

- *query*, utilizzato, nel caso in esame, per specificare un *hashtag*;

- *twittapi*, ossia l'oggetto *api*, senza il quale è impossibile eseguire lo *streaming*, ottenuto a seguito dell'autenticazione presso un *server* di *Twitter* mediante l'utilizzo della libreria *Tweepy*.

Il processo di raccolta dati viene eseguito mediante l'invocazione del metodo *computeRetweets(path)* su un oggetto della classe *TwittersRetweets* che ha come attributi proprio i parametri di ricerca (i.e. *since, until, query*) e l'oggetto *twittapi*. L'esecuzione del metodo *computeRetweets(path)* si articola in tre fasi, come è possibile evincere dalla figura 3.1. Più precisamente:

1. Il metodo *computeRetweets(path)* innanzittutto si occupa di recuperare tutti i *tweets* che soddisfano i parametri di ricerca e gli utenti che li hanno creati. Per fare questo, viste le limitazioni temporali che impone l'*Api* ufficiale di *Twitter*, si rende necessario l'utilizzo della libreria *GetOldTweets*: vengono specificati i criteri di ricerca dei *tweets* da recuperare ed infine il *TweetManager* si occupa di cercarli nella pagina *html* di *Twitter*, per poi restituirli. I *tweets* restituiti sono caratterizzati da due importanti parametri: *tweetid*, ossia l'identificativo univoco del *tweet*, e lo *username*, ossia l'utente che l'ha emesso. Per mezzo di questi parametri, in questa fase vengono costruiti due oggetti, ossia:

    - *dictioTwitters*: un dizionario che ha come chiavi gli *usernames* degli utenti che hanno emesso i *tweets* recuperati e come valori dei dizionari della forma *{'tweetcount' : x}*, dove x è il numero dei *tweets* recuperati (e che quindi soddisfano i criteri di ricerca) che sono attribuibili ad un certo *username*;

- *tweetids*: una lista che ha come elementi dei dizionari della forma *{tweetid : tweetuser}*, dove *tweetid* identifica un certo *tweet* e *tweetuser* identifica l'utente che l'ha emesso.

Questi oggetti vengono opportunamente popolati e poi forniti come *input* alla fase successiva.

2. La fase *2* si occupa di scansionare la lista *tweetids*, restituita dalla fase precedente, per individuare tutti i *retweets* emessi nei confronti dei *tweets* che soddisfano i criteri di ricerca. A fine scansione, questa fase restituisce il dizionario *dictioRetweets*, che ha come chiavi delle tuple del tipo *(retweetuser,tweetuser)*, dove *retweetuser* è lo *username* di un utente che ha *retweettato* almeno un *tweet* emesso dall'utente identificato da *tweetuser*, e come valori dei dizionari del tipo *{'retweetcount' : x}*, dove x è il numero di volte che l'utente *retweetuser* ha *retweettato* dei contenuti dell'utente *tweetuser*.
   Questa volta, tuttavia, tali *retweets* sono ottenuti per mezzo dell'*Api* ufficiale di *Twitter*. In particolare, per ogni elemento *{tweetid : tweetuser}* della lista *tweetids*:

   (a) mediante l'invocazione del metodo *retweets*, messo a disposizione dalla *Twitter Api*, vengono recuperati tutti i *retweets* emessi nei confronti del *tweet* identificato dal *tweetid*. Tali *retweets* vengono restituiti sotto forma di una lista di *status objects*, un formato particolare che viene gestito dalla *Twitter Api*;

   (b) per ogni *status object*, che corrisponde ad un particolare *retweet*, viene recuperato il *JSON* che lo descrive, da cui viene estratto lo *username* dell'utente che ha effettuato il *retweet* stesso (accedendo opportunamente ai campi del *JSON*), che chiamiamo *retweetuser*. Se la tupla *(retweetuser,tweetuser)* esiste già come chiave nel dizionario *dictioRetweets* allora

viene semplicemente aggiornato il valore del campo *'retweetcount'* corrispondente, altrimenti tale tupla viene aggiunta al dizionario come sua nuova chiave con valore *{'retweetcount' : 1}*.
Infine, se lo *username retweetuser* non è presente come chiave nel dizionario *dictioTwitters*, viene aggiunta la nuova chiave *retweetuser* con valore *{'tweetcount' : 0}*, in quanto *retweetuser* non ha mai emesso *tweets* contenenti l'*hashtag* specificato.

Il blocco di codice che implementa le azioni descritte nei precedenti punti (a) e (b) è opportunamente gestito da una clausola *try*: in effetti, come precedentemente detto, l'invocazione del metodo *retweets* della *Twitter Api*, in caso di superamento della soglia di 100 richieste ogni 15 minuti, potrebbe causare il sollevamento dell'eccezione *tweepy.error.RateLimitError*. Tale eccezione verrebbe gestita nella clausola *except* immediatamente successiva, mettendo in pausa il processo di acquisizione dei *retweets* per 15 minuti, in modo tale da non incorrere al blocco di un'ora delle credenziali;

3. Acquisite le informazioni necessarie alla costruzione del *retweet graph* corrispondente alla *query* (ossia all'*hashtag*) ed all'intervallo temporale *since-until* forniti in input, è possibile procedere con il loro salvataggio in memoria. Infatti, giunto a questa fase, il metodo *computeRetweets(path)* si occupa di rendere persistenti i dati collezionati nel dizionario *dictioRetweets*. Innanzitutto apre il file di testo corrispondente al *path* fornito in input e successivamente, per ogni chiave *key* in *dictioRetweets*, si occupa di salvarvi le informazioni nel formato *"key[0],key[1],retweetcount"*, andando a capo per ogni *entry* inserita. Ricordiamo che *key[0]* è *retweetuser*, *key[1]* è *tweetuser* e *retweetcount* è il numero di volte che *retweetuser* ha *retweettato tweets* di *tweetuser*.
In figura 3.2 è possibile osservare un esempio di tale file di testo.

Bisogna sottolineare che questo formato di salvataggio dei dati è stato scelto anche per motivi di compatibilità con lo schema utilizzato dall'articolo [4], in modo tale da poterne riprodurre facilmente i test.

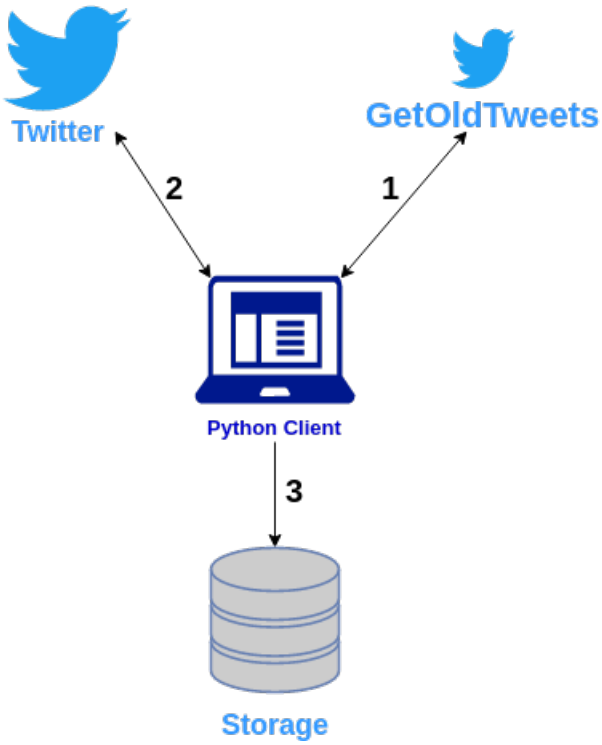


Figura 3.1: Processo di raccolta dati.

```
karlfrisch,castonchris,2.0
karlfrisch,servantonice,2.0
karlfrisch,hattieknuff,2.0
karlfrisch,dgjewel,2.0
rtv6,marcmullins1,3.0
rtv6,ericrtv6,2.0
```

Figura 3.2: Esempio di file che descrive un *retweet graph*.

3.2 Implementazione

In questo paragrafo verrà descritta l'implementazione di tutte quelle fasi che sono successive alla raccolta dati. Fissato un certo *hashtag* che identifica una discussione che ha luogo su *Twitter* ed un intervallo di osservazione, i dati vengono raccolti come illustrato nel paragrafo precedente, per poi essere convertiti in un vero e proprio *retweet graph*, per il quale è possibile:

1. rilevere le *echo-chambers* e calcolare l'*RWC*;

2. eseguire i due algoritmi di *k-edge recommendation*.

In figura 3.3 è sintetizzato tutto il processo al quale viene sottoposto il grafo, che infine conduce ai risultati derivanti dall'applicazione dei due algoritmi alternativi proposti.

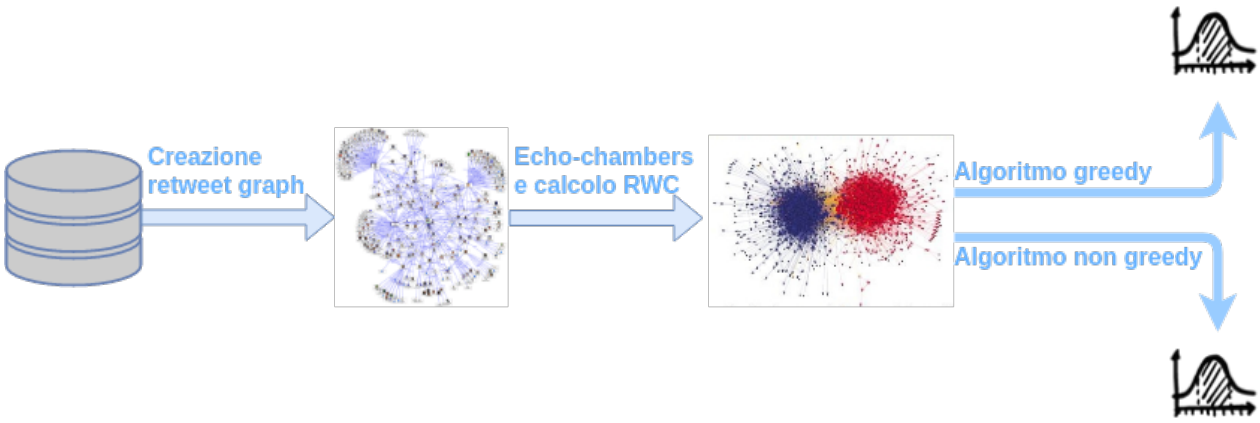


Figura 3.3: *Pipeline* implementativa.

Nel seguito della trattazione verranno illustrati tutti i dettagli implementativi delle fasi di cui si compone la *pipeline* in figura, iniziando dalla fase di creazione del *retweet graph*.

3.2.1 Creazione del *retweet graph*

Per eseguire agevolmente tutte le operazioni di manipolazione ed analisi dei grafi utili al raggiungimento degli obiettivi sopra menzionati, è necessario rappresentare i *retweet graphs* con un formato facilmente gestibile dal linguaggio *Python*. La fase di raccolta dati, illustrata nel paragrafo precedente, al termine della sua esecuzione si preoccupa di rendere persistenti le informazioni raccolte. Tali informazioni sono certamente sufficienti a descrivere il *retweet graph* a cui si riferiscono ma sono caratterizzate da un formato non facilmente manipolabile via codice: occorre rappresentare tali informazioni mediante una struttura dati.
A tale scopo, la classe *EndorsementGraph*, che ha come attributi il *path* del file di testo in input ed il suo nome, permette di:

1. *parsare* il contenuto del file di testo in input, creato nella fase di raccolta dati, per ottenere una struttura dati che rappresenti il *retweet graph* corrispondente;

2. *serializzare* la struttura dati creata e rendere persistente il risultato della serializzazione.

In sintesi, la classe *EndorsementGraph* permette di modellare un *retweet graph*, corrispondente a determinati criteri di ricerca *(querySearch,since,until)* e le cui informazioni sono state già raccolte ed organizzate in un file, con una struttura dati direttamente manipolabile da codice. Il *core* della classe *EndorsementGraph* è il metodo *buildEGraph*, il quale si occupa di costruire tale struttura dati e di serializzarla, per poi memorizzare il risultato della serializzazione nella *directory* il cui *path* è specificato come parametro.
Il metodo *buildEGraph* demanda al *package NetworkX* la creazione della struttura dati che descrive il *retweet graph* considerato e la sua manipolazione. In particolare, *NetworkX* è un pacchetto *Python* per la creazione, la manipolazione e lo studio della struttura, delle dinamiche e delle funzioni di reti complesse. Tra le altre *features*, in

particolare offre strumenti per lo studio della struttura e della dinamica delle reti sociali, biologiche e infrastrutturali, strumenti che sono stati largamente sfruttati nel lavoro di tesi proposto.
Scendiamo ora nei dettagli implementativi del metodo *buildEGraph(destination_dir)*. Esso viene invocato su un oggetto della classe *EndorsementGraph*, caratterizzato dagli attributi *input_dir* e *input_name*, ossia, rispettivamente, la *directory* e il nome del file di testo che descrive il *retweet graph* in considerazione, file creato nella fase di raccolta dati.
Il metodo *buildEGraph* costruisce un *DiGraph* (i.e. grafo diretto) di *NetworkX* a partire dalle informazioni collezionate nel file *input_name* attraverso i seguenti passi:

1. Crea tre dizionari:

    - *dictio_nodes_convert*, in cui ogni chiave è l'*username* di un utente che ha partecipato alla discussione ed il valore corrispondente è un suo identificativo univoco;
    - *dictio_nodes*, in cui ogni chiave è l'*username* di un utente che ha partecipato alla discussione ed il valore corrispondente è il numero di volte che tale utente ha effettuato un *retweet* sul *topic* in questione;
    - *dictio_edges*, in cui ogni chiave identifica una relazione di *endorsement (source,dest)*, dove *source e dest* sono due *usernames*, ed il valore corrispondente è il numero di volte che l'utente *source* ha effettuato un *retweet* nei confronti dell'utente *dest*.

2. Popola tali dizionari *parsando* il file *input_name*, riga per riga, dove ogni riga ha la forma *"source,dest,retweetcount"*, in modo tale che, al termine del *parsing*:

    - sia possibile associare ad ogni *username* di utenti che hanno partecipato alla discussione un identificativo, ossia *dictio_nodes_convert[username]*;

- sia possibile associare ad ogni arco *(source,dest)* una tupla di identificativi *(dictio_nodes_convert[source],dictio_nodes_convert[dest])*;

- sia possibile associare ad ogni tupla *(dictio_nodes_convert[source], dictio_nodes_convert[dest])* una probabilità di *retweet* pari a:

$$P[(source, dest)] = \frac{dictio\_edges[(source, dest)]}{dictio\_nodes[source]} \tag{3.1}$$

3. Utilizzando le informazioni ottenute al passo precedente, il metodo *buildE-Graph* popola il *DiGraph* con dei nodi, identificati da un intero univoco e che hanno come attributi il proprio *retweetcount* (i.e. il numero di volte che l'utente corrispondente al nodo ha effettuato un *retweet* sul *topic* in questione) ed il proprio *username*, e degli archi diretti, che hanno come unico attributo una probabilità di *retweet*, ricavata dalla formula (3.1);

4. Serializza il *DiGraph* ottenuto mediante l'utilizzo del modulo *Pickle*, il quale implementa i protocolli binari per la serializzazione e la de-serializzazione di una struttura di oggetti Python. Infine si preoccupa di rendere persistente il risultato della serializzazione: ciò consente di riutilizzare in futuro il *retweet graph* creato, evitando di dover rieseguire il *parsing*, semplicemente invocando l'operazione di *unpickling* sulla versione serializzata, ottenendo nuovamente l'oggetto originario.

Bisogna sottolineare il fatto che la probabilità di *retweet* (3.1), seppur presente come attributo degli archi del *DiGraph*, non è stata utilizzata nell'implementazione proposta ma è stata comunque inserita per permettere futuri sviluppi del *framework*.

3.2.2 Individuazione delle *echo-chambers*

Il metodo *computeData* del modulo *utilities* si occupa di individuare le *echo-chambers* del *retweet graph* in input, le quali sono indispensabili per ricavare tutti i componenti necessari al calcolo del *random-walk controversy score* ed all'esecuzione degli algoritmi di *k-edge recommendation*.
Come detto nel capitolo precedente, le *echo-chambers* vengono individuate mediante l'esecuzione dell'algoritmo di *Girvan-Newman*, la cui implementazione è resa disponibile dal *package NetworkX*. Tale implementazione richiede di convertire il grafo in input nella sua versione *non diretta* (i.e. grafo *non orientato*), ma nulla vieta in futuro di estendere il codice in modo tale che riesca a gestire anche grafi diretti. Il risultato dell'esecuzione dell'algoritmo di *Girvan-Newman* viene gestito dal metodo *computeData* con l'ausilio dei seguenti dizionari:

- *communities*, in cui ogni chiave corrisponde all'indice di una delle comunità rilevate dall'algoritmo ed il valore corrispondente altro non è che la lista dei nodi del grafo che appartengono a tale comunità;
- *partitions*, in cui ogni chiave è un nodo del grafo ed il valore corrispondente è l'indice della comunità a cui tale nodo appartiene.

I dizionari *communities e partitions* permettono di partizionare perfettamente l'insieme dei nodi del grafo, in modo tale che ciascuno di essi venga associato all'*echo-chamber* a cui appartiene, ossia al lato della *controversia* per il quale si schiera. Nei paragrafi a seguire, sarà possibile notare quanto tali dizionari siano largamente usati ed indispensabili per l'esecuzione del sistema proposto.
È necessario precisare che la scelta dell'algoritmo di *Girvan-Newman* non è stata immediata ma, al contrario, precedentemente è stato valutato un algoritmo di *community detection* alternativo, ossia l'algoritmo di *Louvain*.
Al contrario dell'algoritmo di *Girvan-Newman*, il metodo di *Louvain* non agisce

rimuovendo ad ogni passo l'arco caratterizzato dall'*edge betweenness* più alta ma è incentrato sull'ottimizzazione della *modularità*, che misura la densità dei collegamenti all'interno delle comunità rispetto ai collegamenti tra le comunità. L'algoritmo inizia assegnando ciascun nodo alla propria comunità e consiste in due fasi:

1. *modularity optimization*: in questa fase, per ciascun nodo l'algoritmo esamina quanto cambierebbe la modularità se il nodo venisse rimosso dalla sua comunità e aggiunto alla comunità di ciascuno dei suoi vicini. Il nodo viene quindi inserito nella comunità in cui viene massimizzato il guadagno in modularità. Questo processo viene ripetuto per ciascuno dei nodi fino a quando non è possibile ottenere ulteriori miglioramenti;

2. *community aggregation*: l'algoritmo ora crea una nuova rete i cui nodi sono le comunità trovate nella prima fase.

Queste fasi vengono ripetute in modo iterativo fino al raggiungimento della modularità massima. In sintesi, è possibile dire che l'approccio di *Louvain* si basa su un processo di *aggregazione* della rete mentre l'approccio di *Girvan-Newman* si basa su un processo di *decomposizione* della rete.
Tuttavia il processo di *aggregazione* di *Louvain* è volto all'ottimizzazione della *modularità* e quindi nel momento in cui si arresta potrebbe aver rilevato un numero di comunità maggiore di due; al contrario *Girvan-Newman*, essendo caratterizzato da un processo di *decomposizione*, può essere arrestato nel momento in cui rileva esattamente due comunità.
Questo è sostanzialmente il motivo che ha condotto alla scelta dell'algoritmo di *Girvan-Newman*, non tralasciando le motivazioni addotte nel precedente capitolo *Teoria alla base del problema ed algoritmi per la risoluzione*.

3.2.3 Calcolo dell'*RWC*

Una volta rilevate le *echo-chambers* del *retweet graph* in input, ossia il *DiGraph* ricavato nella precedente fase di *"creazione del retweet graph"*, è possibile ottenere tutte le componenti utili al calcolo dell'*indice di controversia RWC*, le quali dipendono proprio dalla struttura delle comunità.
Il processo di calcolo dell'*RWC* del grafo in input è illustrato in figura 3.4.

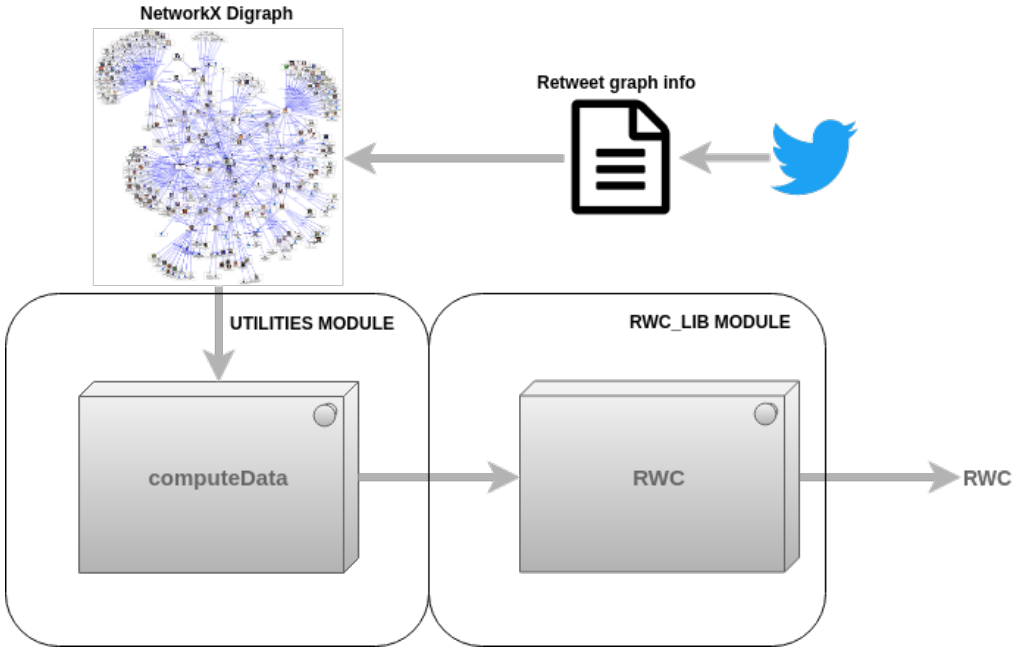


Figura 3.4: Processo di calcolo del *random-walk controversy score* di un *retweet graph*.

Come illustrato in figura, il *DiGraph di NetworkX* ottenuto nella fase di *"creazione del retweet graph"* viene fornito come input al metodo *computeData* il quale, dopo averne individuato le *echo-chambers*, produce[1]:

[1]Oltre ai dati elencati, il metodo *computeData* produce anche i dizionari *communities,partitions* e le liste dei nodi delle due comunità *sorted_x,sorted_y*, ordinate secondo l'*in degree* dei nodi stessi. Ad ogni modo questi dati non sono utili al calcolo dell'*RWC* e pertanto per il momento non verranno considerati.

- i vettori di *restart* e_x ed e_y, dove X, Y indicano le due *echo-chambers* del *DiGraph*;
- i vettori c_x e c_y;
- le matrici di transizione P_x e P_y e quindi le matrici M_x^{-1} ed M_y^{-1}.

Per chiarimenti riguardo il significato di questi vettori e matrici si consiglia la lettura del capitolo precedente.
Il metodo *rwc* del modulo *rwc_lib* ha il compito di utilizzare questi dati per calcolare finalmente il *random-walk controversy score* del *DiGraph* in input, applicando la seguente espressione:

$$RWC(G, X, Y) = (1 - \alpha)(c_x - c_y)^{\mathrm{T}}(M_x^{-1}e_x - M_y^{-1}e_y) \tag{3.2}$$

Dove α, fornito come parametro al metodo *rwc*, è la *probabilità di continuare il random-walk* e quindi $(1 - \alpha)$ è la *restart probability*.
Come già precedentemente affermato, i due algoritmi alternativi di *k-edge recommendation* sono stati implementati perseguendo anche l'obiettivo di limitare l'invocazione del metodo *computeData*, poiché esso presenta un peso computazionale piuttosto ingente dovuto, tra le altre cose, al calcolo delle matrici inverse M_x^{-1} ed M_y^{-1}.
L'esecuzione di entrambi gli algoritmi di *k-edge recommendation* proposti richiede di calcolare, per ogni arco scansionato, il δRWC che esso apporterebbe se fosse aggiunto al grafo. Se non si utilizzasse una strategia di calcolo più efficiente, per ottenere tale δRWC bisognerebbe misurare l'RWC del grafo a seguito dell'aggiunta dell'arco in considerazione, per poi effettuare la differenza con l'RWC prima dell'aggiunta. Ciò renderebbe necessaria l'invocazione del metodo *computeData* per un numero di volte pari al numero di archi scansionati, con l'effetto di un degrado prestazionale

non trascurabile.

Con l'obiettivo di ovviare a tutto ciò, l'implementazione proposta di entrambi gli algoritmi di *k-edge recommendation* utilizza la tecnica di *Sherman-Morrison*, già illustrata nel capitolo precedente, la quale permette di evitare di invocare continuamente il metodo *computeData* per il calcolo dei δRWC.

3.2.4 Implementazione degli algoritmi proposti

Entrambi gli algoritmi di *k-edge recommendation* implementati, ossia *non greedy* e *greedy*, scelgono i k archi più promettenti, in termini del decremento dell'*RWC* che consentirebbero se fossero aggiunti al grafo, partendo dallo stesso dominio. Tale dominio è costituito da tutti gli archi diretti, non ancora materializzati nel *retweet graph* considerato, che connettono i k_1 vertici con *in degree* più alto dell'*echo-chamber* X con i k_2 vertici con *in degree* più alto dell'*echo-chamber* Y e viceversa, dove k_1 e k_2 devono essere forniti come input.

I due algoritmi, come precedentemente affermato, differiscono tra loro solo per la modalità di selezione dei k archi: questa differenza è facilmente deducibile dallo *pseudo-codice* dei metodi che li implementano.

Algorithm 2 *non_greedy_alg(g, data, X, Y, α, k_1, k_2, k)*

```
sorted_x ← sort_nodes(g, X);
sorted_y ← sort_nodes(g, Y);
domain_edges ← get_domain_edges(g, α, k1, k2, sorted_x, sorted_y, data);
sorted_edges ← sort_by_delta_rwc(domain_edges);
best_edges ← sorted_edges[0:k];
return best_edges;
```

Osservando *Algorithm 2 e Algorithm 3*, risulta subito evidente la minore complessità dello *pseudo-codice* dell'algoritmo *non-greedy* che, vedremo nei test, corrisponde ad una maggiore efficienza per quanto concerne i tempi di esecuzione ma anche ad una minore efficacia per quanto concerne il decremento dell'*RWC*, rispetto a ciò che

invece consente l'algoritmo *greedy*.
Poniamo ora l'attenzione sullo *pseudo-codice* dell'algoritmo *non-greedy* e cerchiamo di analizzarlo, iniziando dai parametri in input:

- g è il *DiGraph* in input, ossia il *retweet graph* (caratterizzato da un certo *hashtag*) di cui si vuole ridurre il *random-walk controversy score*;
- *data* è l'insieme di vettori e matrici restituiti dal metodo *computeData*, come osservato precedentemente;
- X, Y sono le due comunità (i.e. *echo-chambers*) del grafo, anch'esse restituite dal metodo *computeData* ed individuate mediante l'esecuzione dell'algoritmo di *Girvan-Newman*;
- α, k_1, k_2 e k sono, rispettivamente, la *probabilità di continuare il random-walk*, il numero di vertici con *in degree* più alto della comunità X da considerare, il numero di vertici con *in degree* più alto della comunità Y da considerare ed il numero di archi da consigliare.

Come è possibile osservare dal suo *pseudo-codice*, la modalità di selezione dei k archi utilizzata dall'algoritmo *non-greedy* si compone dei seguenti passi:

1. *sorting* dei nodi della comunità X e *sorting* dei nodi della comunità Y rispetto al loro *in degree*;
2. costruzione del dominio dei $2 \times k_1 \times k_2$ archi diretti che collegano i k_1 vertici con *in degree* più alto dell'*echo-chamber* X con i k_2 vertici con *in degree* più alto dell'*echo-chamber* Y e viceversa, tra i quali verranno scelti i k più promettenti per la riduzione dell'*RWC*. Bisogna sottolineare che tale dominio potrebbe contenere meno di $2 \times k_1 \times k_2$ archi, qualora qualcuno di essi fosse già presente nel grafo. Nello *pseudo-codice* questo passo viene espletato

da *get_domain_edges*, alla cui implementazione è richiesto anche di associare ad ogni arco del dominio il proprio δRWC, calcolato mediante la tecnica di *Sherman-Morrison*;

3. *sorting* degli archi del dominio individuato al punto precedente in ordine decrescente dei loro δRWC, cambiati di segno. Nello *pseudo-codice* questo passo viene espletato da *sort_by_delta_rwc*;

4. *return* dei *top-k* archi dell'insieme ordinato al punto precedente, ossia i *best_edges*.

Algorithm 3 *greedy_alg(g, data, X, Y, α, k_1, k_2, k)*

```
Initialization : best_edges ← [];
for i = 0:k do
  sorted_x ← sort_nodes(g, X);
  sorted_y ← sort_nodes(g, Y);
  domain_edges ← get_domain_edges(g, α, k1, k2, sorted_x, sorted_y, data);
  sorted_edges ← sort_by_delta_rwc(domain_edges);
  g.add_edge(sorted_edges[0]);
  data ← compute_data(g, α, X, Y);
  best_edges.append(sorted_edges[0]);
end for
return best_edges;
```

La versione *greedy* dell'algoritmo di *k-edge recommendation* è caratterizzata dagli stessi parametri in input della versione *non-greedy* ma, come si osserva dal suo *pseudo-codice*, differisce da quest'ultima per quanto riguarda la modalità di scelta dei k archi da proporre. In particolare, si nota che in ognuno dei k passi impiegati per individuare i k archi esegue le seguenti operazioni:

1. *sorting* dei nodi della comunità X e *sorting* dei nodi della comunità Y rispetto al loro *in degree*;

2. costruzione del dominio dei $2 \times k_1 \times k_2$ archi diretti, come descritto al punto 2 dell'algoritmo *non-greedy*;

3. *sorting* degli archi del dominio individuato al punto precedente, come descritto al punto 3 dell'algoritmo *non-greedy*;

4. aggiunta al grafo g dell'arco diretto migliore, individuato al punto precedente. Per migliore si intende in termini del δRWC associato. Nello *pseudo-codice* questo passo viene espletato da *add_edge*;

5. ricalcolo dell'oggetto *data*, mediante l'invocazione del metodo *computeData*, necessario a seguito dell'aggiunta del nuovo arco al grafo g. Come è possibile notare dallo *pseudo-codice*, in questo caso al metodo *computeData* vengono passate come parametri anche le due *echo-chambers* X, Y. L'algoritmo *greedy*, infatti, assume che le due comunità X, Y restino invariate a seguito delle aggiunte degli archi al grafo: per permettere questo, l'implementazione di *computeData* consente di scegliere se specificare o meno le *echo-chambers* come parametri in input e, in caso vengano specificate, esse non vengono ricalcolate nuovamente.

   Oltretutto, l'esigenza dell'algoritmo *greedy* di fissare le *echo-chambers* consente di ottenere un vantaggio prestazionale, visto che permette di non eseguire continuamente l'algoritmo di *Girvan-Newman*[2], il quale risulta oneroso soprattutto per grafi con molti nodi ed archi, come nel caso in esame;

6. aggiornamento della lista *best_edges* mediante l'aggiunta dell'arco diretto individuato al punto 3 (i.e. *sorted_edges[0]*).

[2]In caso contrario, sarebbe stato necessario invocare l'algoritmo di *Girvan-Newman* un numero di volte pari a k, con l'effetto di un degrado prestazionale inaccettabile.

Al termine del *loop*, l'algoritmo *greedy* restituisce la lista dei k archi costruita in k passi, ovvero la lista *best_edges*.
Come già osservato precedentemente, l'algoritmo *greedy* è senz'altro più lento dell'algoritmo *non-greedy* di un fattore k, visto che impiega k passi per scegliere k archi. Questa caratteristica, tuttavia, fa sì che la modalità di selezione degli archi adottata dalla versione *greedy* sia più precisa e permetta, a parità di archi aggiunti, un maggiore decremento del *random-walk controversy score* del grafo rispetto a quanto permette la versione *non-greedy*. Il capitolo relativo ai test dimostrerà l'attendibilità di queste osservazioni.

3.2.5 Strumento per la visualizzazione degli archi consigliati

Sarebbe interessante confrontare i due algoritmi di *k-edge recommendation* non solo dal punto di vista del decremento dell'*RWC* che, a parità di archi proposti, consentono ma anche dal punto di vista delle caratteristiche dei nodi estremi (i.e. *endpoints*) dei k archi che scelgono. Infatti, poiché i due algoritmi differiscono per la modalità di scelta dei k archi, anche i nodi (ossia gli utenti del *social network*) che essi coinvolgono saranno, in generale, diversi.
Pertanto la visualizzazione di tali nodi, e di alcune loro caratteristiche tra cui l'*in-degree*, l'*out-degree* e l'*username* dell'utente al quale corrispondono, potrebbe essere uno strumento ausiliario per individuare le cause che fanno sì che un algoritmo sia più efficace dell'altro.
A tale scopo, il sistema proposto implementa un *tool* di visualizzazione dei k archi scelti e dei loro nodi estremi, descritti dal corrispondente *username* e dal loro *in* e *out-degree*. In particolare, il *tool* ha l'obiettivo di evidenziare i k archi risultanti dal processo di selezione dell'algoritmo di *recommendation* considerato e di filtrare tutti i nodi del grafo che non sono *endpoints* di nessuno di tali archi. A tal proposito, il *tool* effettua una sorta di *differenza* tra:

1. il *retweet graph* a seguito dell'esecuzione dell'algoritmo di *recommendation*[3];

2. il *retweet graph* prima dell'esecuzione dell'algoritmo di *recommendation*.

Pertanto, il grafo differenza, se si considerano solo i k archi proposti, rientra nella classe dei *grafi bipartiti*[4], in quanto è possibile partizionare l'insieme dei suoi nodi in due sottoinsiemi tali che ogni nodo di una di queste due parti è collegato solo a nodi dell'altra. Potendo trattare il grafo differenza come un grafo bipartito, è possibile *colorare* tutti i suoi nodi con soli due colori, in modo tale che non esista nessuna coppia di nodi adiacenti caratterizzati dallo stesso colore.
In sintesi, quindi, il *tool* proposto opera mediante i seguenti passi:

1. ricava il grafo differenza appena descritto;

2. *colora* i nodi del grafo differenza, sfruttando la struttura bipartita, ed evidenzia i k archi proposti;

3. per ogni nodo del grafo differenza, inserisce lo *username* corrispondente, il suo *in-degree* ed *out-degree*.

Il *coloring* dei nodi del grafo differenza viene eseguito dal metodo di *NetworkX* *bipartite.color(bipartite_graph)*, dove *bipartite_graph* è un grafo diretto i cui archi sono tutti e soli i k archi scelti dall'algoritmo di *recommendation* considerato ed i cui nodi sono tutti e soli i nodi che sono *endpoints* di almeno uno tra tali archi.
Evitando di scendere troppo nei dettagli implementativi, inseriamo di seguito un esempio di *output* tipico del *tool* proposto, che aiuta a chiarire quanto già detto (figura 3.5).

[3]Si ipotizza che tutti i k archi consigliati vengano accettati e che quindi compaiano nel grafo. Quest'assunzione verrebbe meno se si considerasse anche la probabilità di accettazione.

[4]Infatti i k archi diretti proposti, per definizione degli algoritmi di *k-edge recommendation*, non possono congiungere nodi appartenenti alla stessa *echo-chamber*.

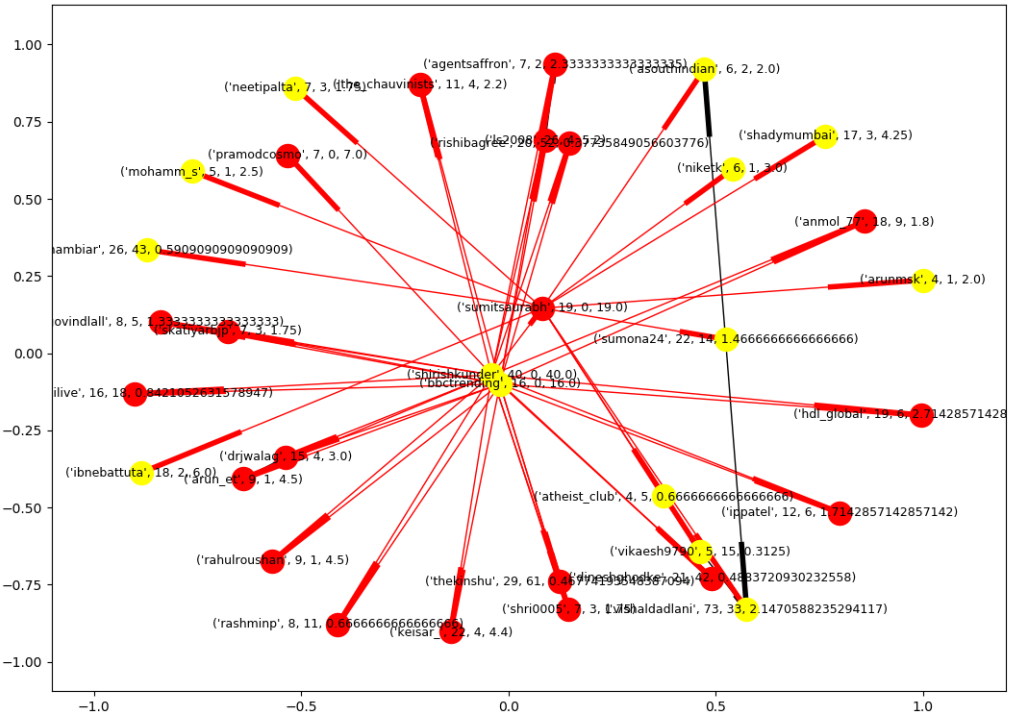


Figura 3.5: *Output* tipico del *tool* di visualizzazione.

I nodi *gialli* appartengono ad una *comunità* ed i nodi *rossi* all'altra; gli archi evidenziati in *rosso* sono i k archi diretti scelti dall'algoritmo di *k-edge recommendation* utilizzato. Si può notare come ad ogni nodo venga associato, nell'ordine, l'*username* corrispondente, il suo *in degree*, *out degree* ed il rapporto $\frac{indegree}{outdegree+1}$.

Capitolo 4

Test dell'*edge-recommendation system* in modalità *greedy e non*

Il sistema implementato è stato sottoposto a vari test con lo scopo di valutare i due algoritmi di *k-edge recommendation* alternativi, ovvero *greedy e non greedy*, ed in modo tale da poterli confrontare tra loro in termini di:

- decremento totale dell'RWC che ciascuno di essi consente di apportare ad un certo *retweet graph* in input, a parità di numero di archi proposti k;
- qualità dei k archi scelti, in termini del δRWC associato a ciascuno di essi;
- tempi di esecuzione.

I *retweet graphs* utilizzati come input dei test corrispondono alle discussioni attorno agli *hashtags* controversi *#beefban, #indiana, #russia_march*, le cui informazioni (i.e. *tweets* e *retweets* emessi nel periodo di osservazione) sono reperibili presso il *repository* degli autori dell'articolo [4]. Pertanto, in questo caso, non è stato necessario eseguire il *processo di raccolta dati* descritto nel capitolo precedente ma, per ciascuno degli *hashtags* appena menzionati, è bastato *parsare* il relativo file dei

retweets (disponibile nel *repository*) e creare il *retweet graph* corrispondente.
I *retweet graphs* creati, relativi agli *hashtags* di cui sopra, hanno le seguenti caratteristiche:

Hashtag	**|V|**	**|E|**
#beefban	1610	1978
#indiana	2467	3143
#russia_march	2134	2951

I parametri del sistema sono stati impostati con i seguenti valori:

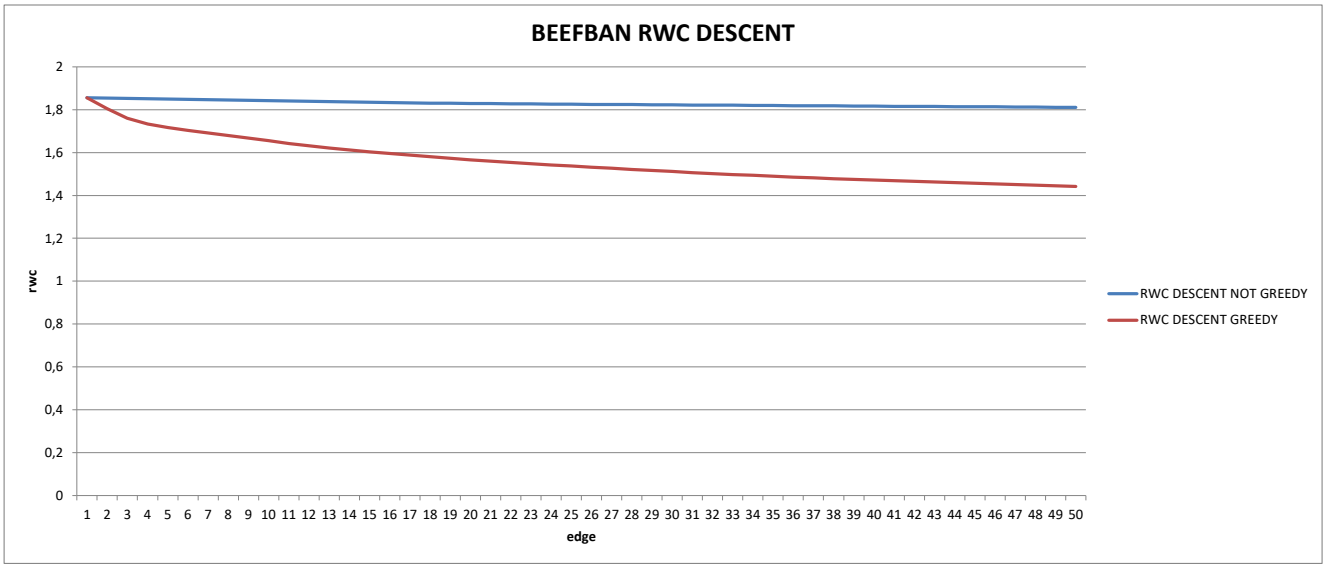
- $\alpha = 0.85$;
- $k_1 = 20$;
- $k_2 = 20$;
- $k = 50$.

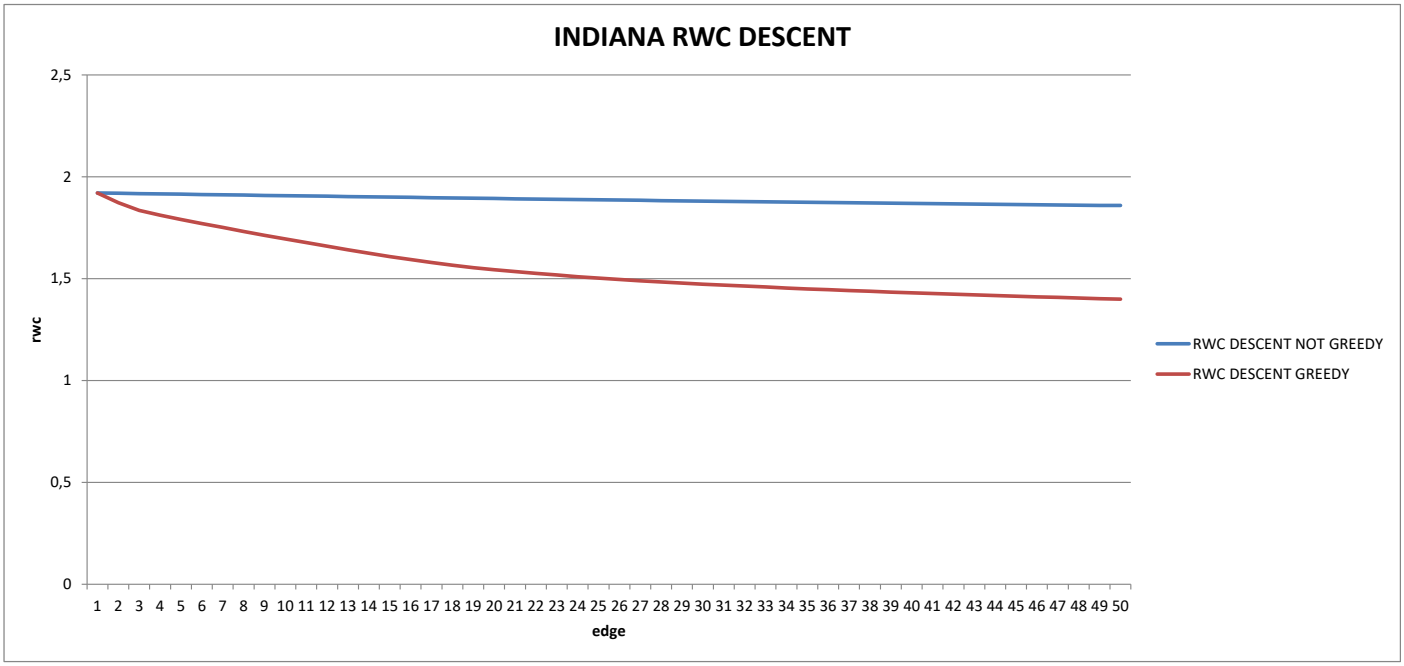
Nel prossimo paragrafo, per ciascuno dei tre *retweet graphs* in input, verranno mostrati e commentati i risultati dei test relativi alla discesa dell'*RWC* ottenuta a seguito dell'applicazione di ciascuno dei due algoritmi di *recommendation* proposti.

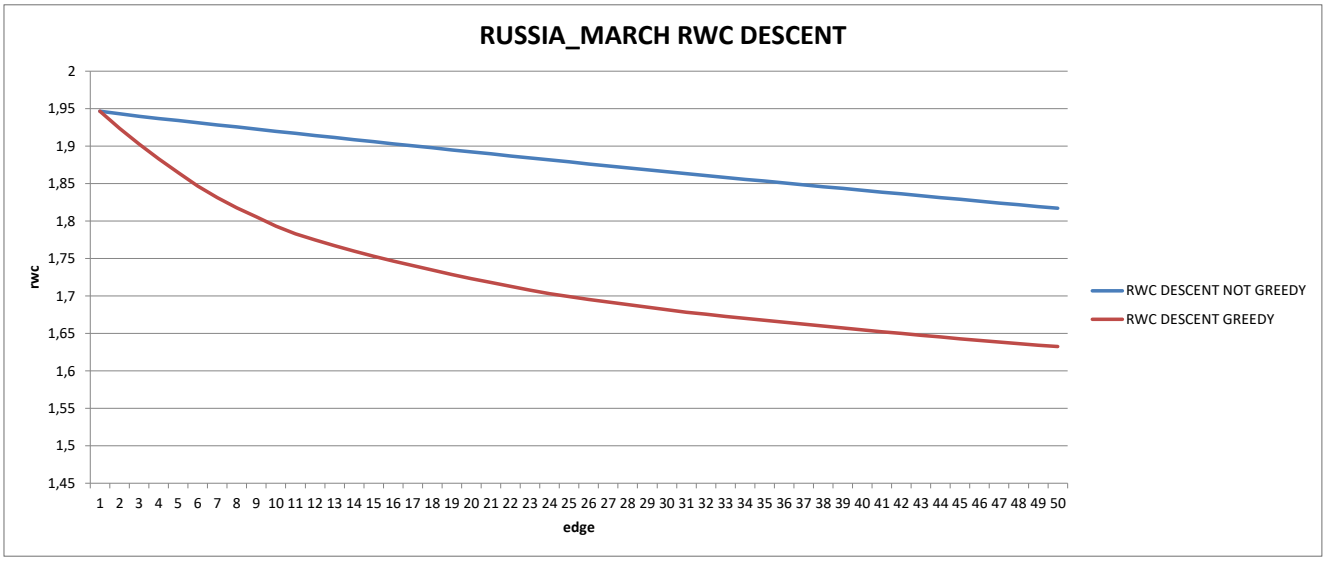
4.1 Discesa dell'*RWC*

Di seguito inseriamo i grafici che mostrano la discesa dell'*RWC* dei *retweet graphs* relativi agli *hashtags* considerati, nell'ordine:

1. *#beefban*;
2. *#indiana*;
3. *#russia_march*.


BEEFBAN RWC DESCENT
2
1,8
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
rwc
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
edge
RWC DESCENT NOT GREEDY
RWC DESCENT GREEDY


INDIANA RWC DESCENT
2,5
2
1,5
1
0,5
0
rwc
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
edge
RWC DESCENT NOT GREEDY
RWC DESCENT GREEDY


RUSSIA_MARCH RWC DESCENT
rwc
2
1,95
1,9
1,85
1,8
1,75
1,7
1,65
1,6
1,55
1,5
1,45
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
edge
RWC DESCENT NOT GREEDY
RWC DESCENT GREEDY

Ciascuno dei tre grafici, uno per *retweet graph*, ha l'obiettivo di porre a confronto i decrementi dell'*RWC* che rispettivamente i due algoritmi di *edge recommendation* consentono di raggiungere a parità di archi proposti.
In particolare, fissato un *retweet graph* g, il grafico corrispondente mostra due funzioni, rispettivamente di colore *rosso* e di colore *blu*, con valori nel dominio $0 < j \leq k$:

- $RWC(g,j)_{greedy}$, ovvero l'*RWC* che caratterizzerebbe il *retweet graph* g qualora i primi j archi proposti dall'algoritmo *greedy* si materializzassero nel grafo;
- $RWC(g,j)_{non\text{-}greedy}$, ovvero l'*RWC* che caratterizzerebbe il *retweet graph* g qualora i primi j archi proposti dall'algoritmo *non-greedy* si materializzassero nel grafo.

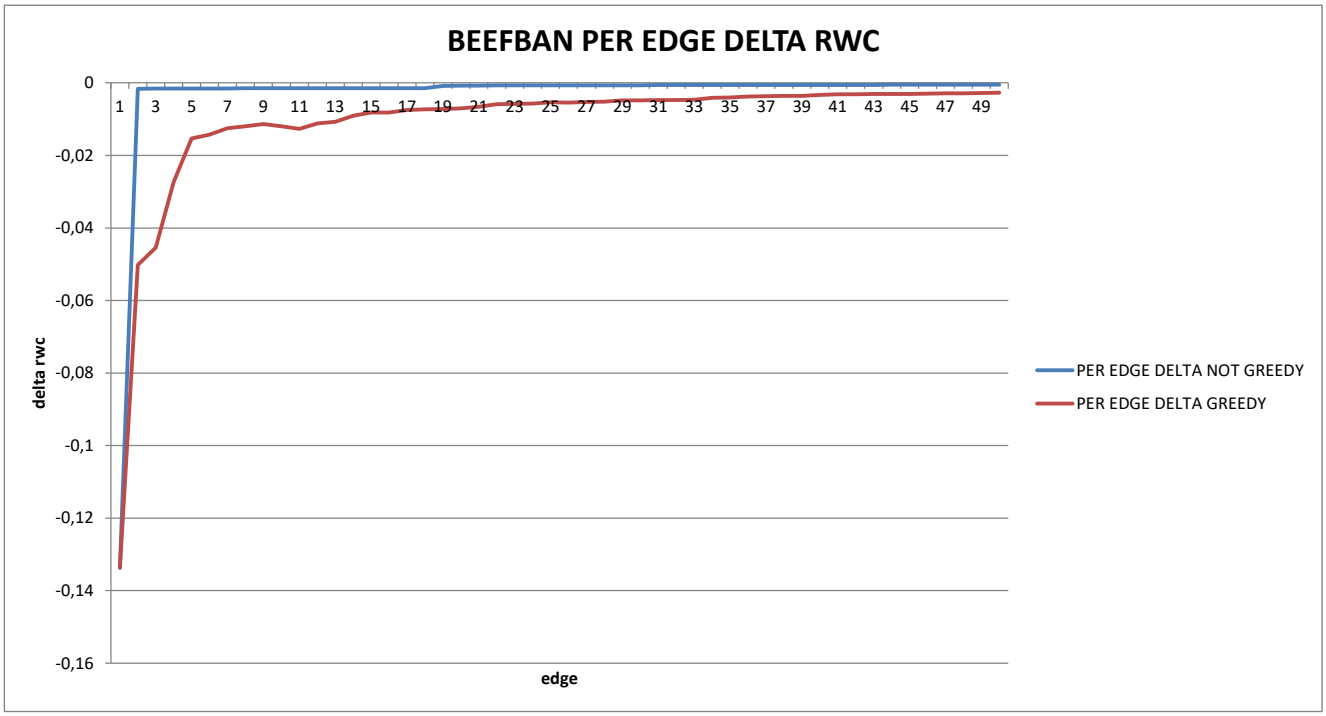
Come si nota immediatamente dai grafici, per ogni *retweet graph* g considerato vale:

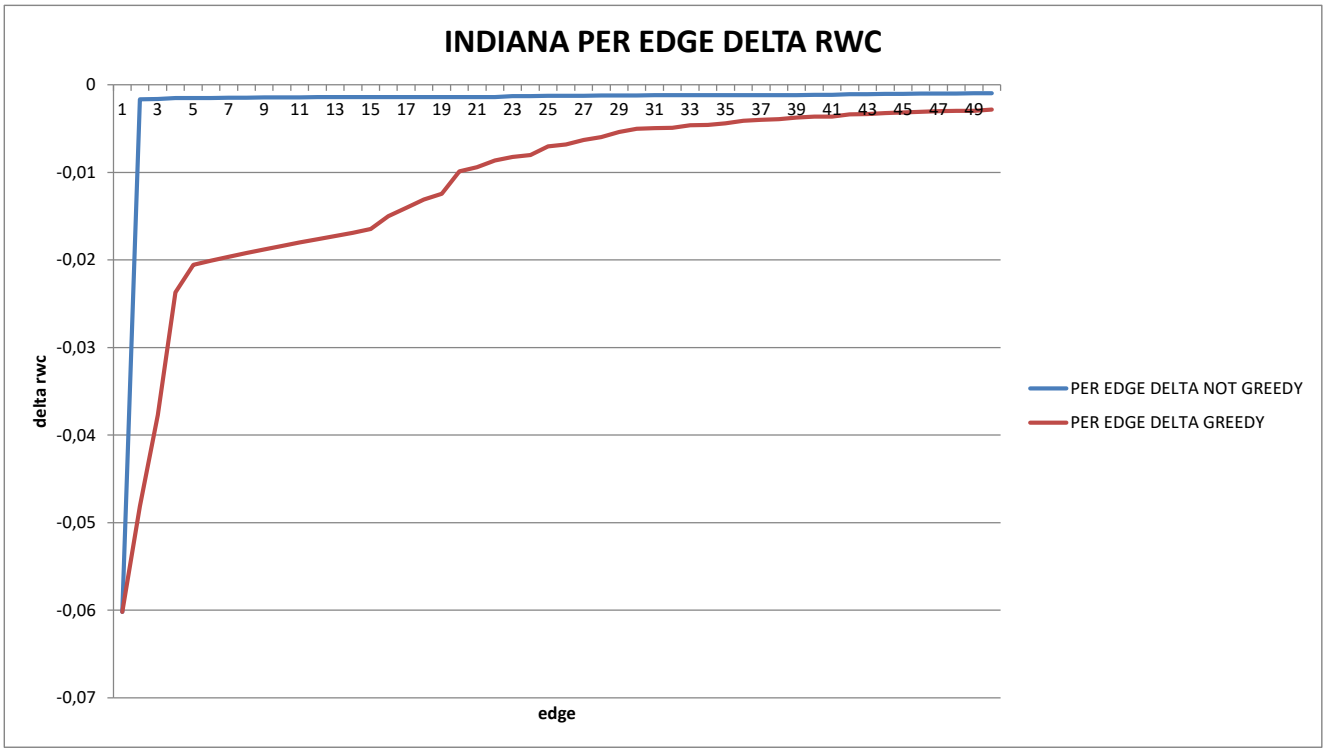
$$RWC(g,j)_{greedy} \leq RWC(g,j)_{non\text{-}greedy}, \forall j = 1, .., k$$

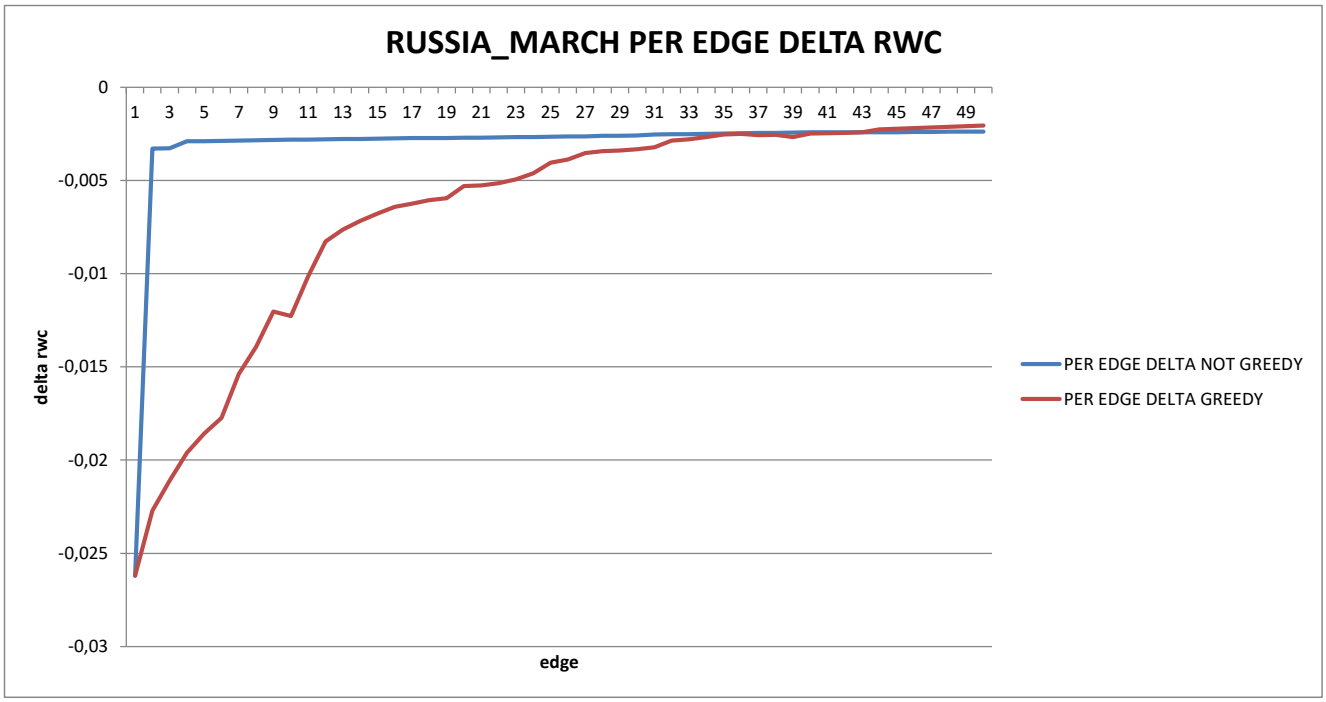
Ovvero, a parità di archi proposti, l'algoritmo *greedy* consente *sempre* di raggiungere un decremento dell'*RWC* maggiore o uguale, per ciascun grafo g. Questa maggiore *efficacia* dell'algoritmo *greedy* non soprende, viste la considerazioni e le analisi condotte nei capitoli precedenti, e deriva sostanzialmente dalla maggiore *qualità*, in termini di decremento del *grado di controversia*, di ciascun arco che propone.
Il paragrafo a seguire si occuperà proprio di confrontare gli archi proposti dai due algoritmi *greedy* e *non-greedy*, in termini dei δRWC corrispondenti.

4.2 Qualità degli archi proposti

Ognuno dei seguenti grafici, corrispondenti ai tre *retweet graphs* considerati, associa ad ogni arco, proposto da ognuno dei due algoritmi, il relativo δRWC.


BEEFBAN PER EDGE DELTA RWC
0
-0,02
-0,04
-0,06
-0,08
-0,1
-0,12
-0,14
-0,16
delta rwc
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49
edge
PER EDGE DELTA NOT GREEDY
PER EDGE DELTA GREEDY


INDIANA PER EDGE DELTA RWC
0
-0,01
-0,02
-0,03
-0,04
-0,05
-0,06
-0,07
delta rwc
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49
edge
PER EDGE DELTA NOT GREEDY
PER EDGE DELTA GREEDY


RUSSIA_MARCH PER EDGE DELTA RWC
0
-0,005
-0,01
-0,015
-0,02
-0,025
-0,03
delta rwc
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49
edge
PER EDGE DELTA NOT GREEDY
PER EDGE DELTA GREEDY

Questa volta ciascuno dei tre grafici, uno per *retweet graph*, ha l'obiettivo di porre a confronto i δRWC associati ai k archi che rispettivamente i due algoritmi di *edge recommendation* propongono.
Più precisamente, fissato un *retweet graph* g, il grafico corrispondente mostra due funzioni, rispettivamente di colore *rosso* e di colore *blu*, con valori nel dominio $0 < i \leq k$[1]:

- $\delta RWC(g,i)_{greedy}$, ovvero il δRWC che l'*i-esimo* arco proposto dall'algoritmo *greedy* consentirebbe di ottenere qualora si materializzasse successivamente a *tutti* gli archi proposti con indici da *1* a *i-1*;
- $\delta RWC(g,i)_{non\text{-}greedy}$[2], ovvero il δRWC che l'*i-esimo* arco proposto dall'algoritmo *non-greedy* consentirebbe di ottenere qualora si materializzasse successivamente a *tutti* gli archi proposti con indici da *1* a *i-1*.

Anche in questo caso è evidente che, per ogni *retweet graph* g considerato e $\forall i = 1, .., k$, vale:

$$\delta RWC(g,i)_{greedy} \leq \delta RWC(g,i)_{non\text{-}greedy}$$

Ovvero, nell'ipotesi che tutti gli archi precedentemente proposti vengano accettati, il δRWC associato all'*i-esimo* arco proposto dall'algoritmo *greedy* è minore o uguale al δRWC associato all'*i-esimo* arco proposto dall'algoritmo *non-greedy*, $\forall i = 1, .., k$.
Quest'osservazione implica che, generalmente, gli archi scelti da *greedy* sono migliori *qualitativamente* rispetto a quelli scelti da *non-greedy*, poiché determinano un maggior decremento dell'RWC associato al *retweet graph* in input.

[1] Ovvero l'*i-esimo* arco proposto.

[2] Ciascun $\delta RWC(g,i)_{non\text{-}greedy}$ non è il δRWC utilizzato dall'algoritmo *non-greedy* come criterio di scelta dell'*i-esimo* arco ma è l'effettivo decremento dell'RWC che l'*i-esimo* arco proposto apporterebbe se si materializzasse secondo l'ordine di scelta.

I risultati sinora ottenuti derivano senz'altro dalla maggior precisione del criterio di scelta degli archi dell'algoritmo *greedy* rispetto a quello dell'algoritmo *non-greedy*.
A tal proposito, è possibile utilizzare il *tool di visualizzazione* introdotto nel capitolo precedente per analizzare, per ogni *retweet graph* in input, le caratteristiche dei k archi scelti da ciascuno dei due algoritmi e dei nodi coinvolti: quest'analisi potrebbe chiarire ulteriormente le cause che rendono un algoritmo più efficace dell'altro.
A fine capitolo inseriamo gli *outputs* del *tool*, ciascuno dei quali descrive i risultati dell'applicazione di uno dei due algoritmi di *edge recommendation* su un *retweet graph* in input.
Con riferimento alle figure da 4.1 a 4.6, si nota che generalmente i k archi scelti dall'algoritmo *non greedy* tendono a formare una struttura a *stella*: infatti la maggior parte di essi condivide uno stesso nodo *endpoint*. Ciò è dovuto al fatto che l'algoritmo *non greedy* utilizza come metrica di scelta degli archi il δRWC che ciascuno di essi apporterebbe se fosse aggiunto *individualmente* al grafo, ignorando la potenziale diminuzione dell'efficacia individuale causata dalla loro interazione reciproca. In generale, si osserva che maggiore è la tendenza degli archi scelti a condividere uno stesso *endpoint* e minore è il decremento dell'RWC che collettivamente riescono ad apportare: questo è spesso il caso degli archi proposti dall'algoritmo *non greedy* e determina il suo *deficit* di efficacia.

4.3 Tempi di esecuzione

Questo paragrafo ha lo scopo di mostrare e commentare i tempi di esecuzione dei due algoritmi di *k-edge recommendation*, espressi in funzione del *retweet graph* in input. Si consideri la seguente tabella:

Hashtag	**greedy**	**non greedy**
#beefban	7252 sec	149 sec
#indiana	17580 sec	352 sec
#russia_march	12720 sec	253 sec

Come era facile intuire, i tempi di esecuzione di entrambi gli algoritmi crescono all'aumentare della complessità del *retweet graph* in input, complessità espressa in termini del numero di nodi e del numero di archi. Addirittura l'algoritmo *greedy* con input il *retweet graph #indiana*, che è il grafo più ingente tra i tre esaminati, necessita di quasi 5 ore di esecuzione.
Il tempo di esecuzione è anche funzione dell'algoritmo di *k-edge recommendation* utilizzato: infatti i test hanno confermato le osservazioni riportate nel capitolo 2, nelle quali si asseriva che, a parità di valori dei parametri di sistema (α, k_1, k_2, k) e del *retweet graph* in input, l'algoritmo *greedy* è circa k volte più lento dell'algoritmo *non-greedy*. Nei casi in esame:

- 7252 sec $\simeq k \times$ 149 sec;
- 17580 sec $\simeq k \times$ 352 sec;
- 12720 sec $\simeq k \times$ 253 sec.

Dove k è il numero di archi da proporre ed in questo caso vale 50.
La maggiore precisione e, quindi, la maggiore efficacia dell'algoritmo *greedy* si paga con il suo tempo di esecuzione, il quale può risultare addirittura proibitivo nel caso di grafi molto ingenti. Pertanto, la scelta dell'algoritmo da utilizzare va effettuata caso per caso e dipende dalle esigenze che si vogliono soddisfare, che riguardino i tempi di esecuzione o l'efficacia degli archi proposti.

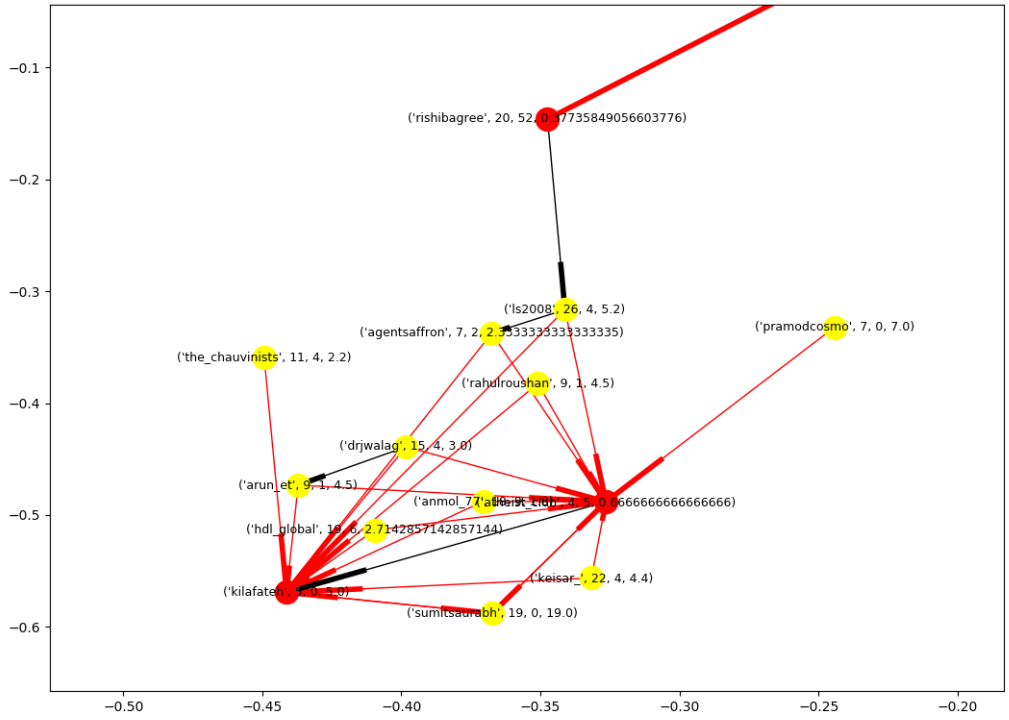


Figura 4.1: Porzione dell'*output* del *tool* a seguito dell'esecuzione di *greedy* sul *retweet graph #beefban*.

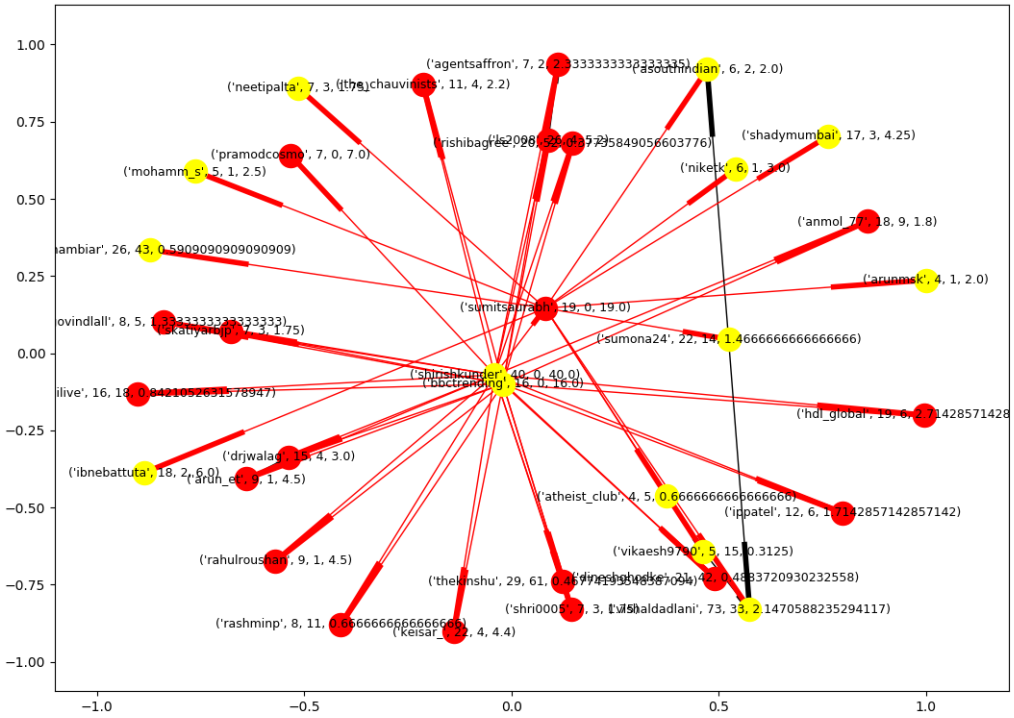


Figura 4.2: Porzione dell'*output* del *tool* a seguito dell'esecuzione di *non greedy* sul *retweet graph #beefban*.

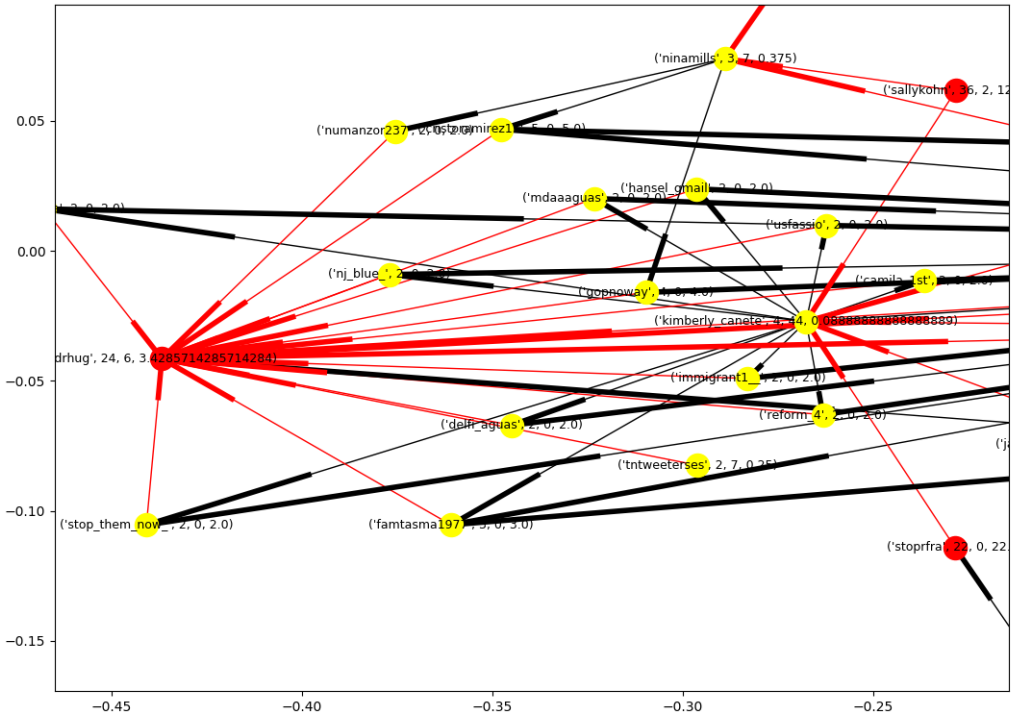


Figura 4.3: Porzione dell'*output* del *tool* a seguito dell'esecuzione di *greedy* sul *retweet graph #indiana*.

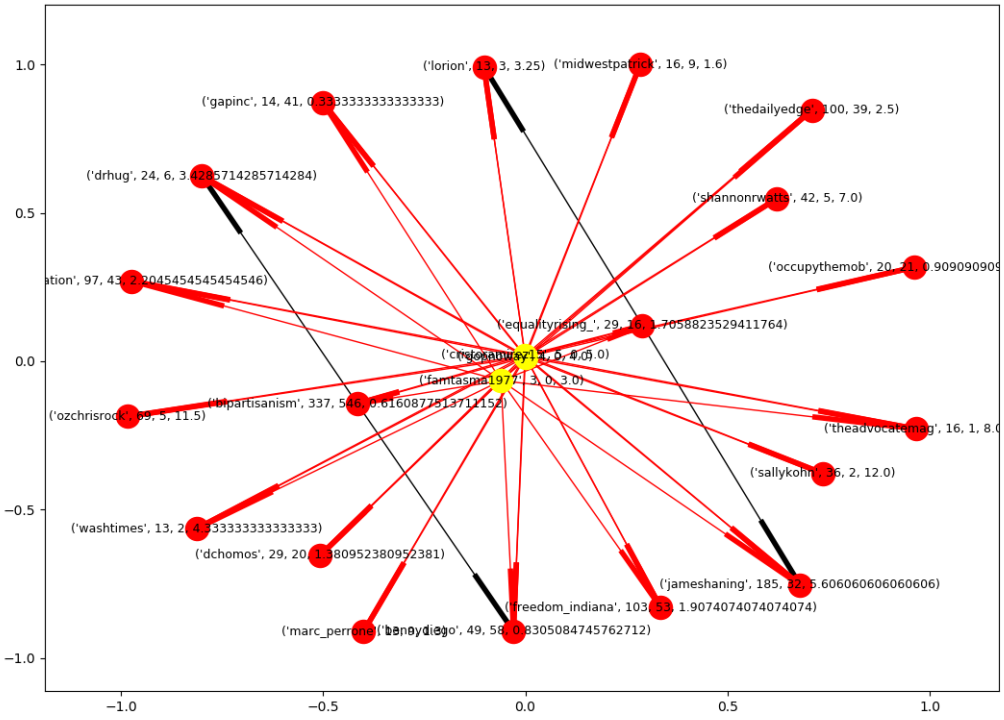


Figura 4.4: Porzione dell'*output* del *tool* a seguito dell'esecuzione di *non greedy* sul *retweet graph #indiana*.

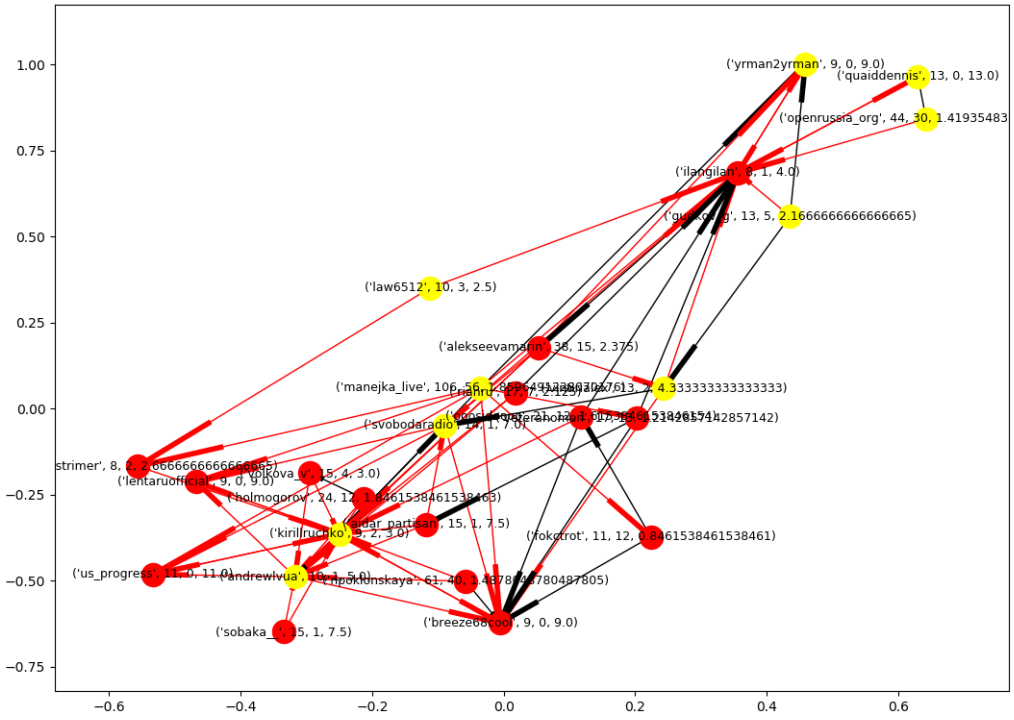


Figura 4.5: Porzione dell'*output* del *tool* a seguito dell'esecuzione di *greedy* sul *retweet graph #russia_march*.

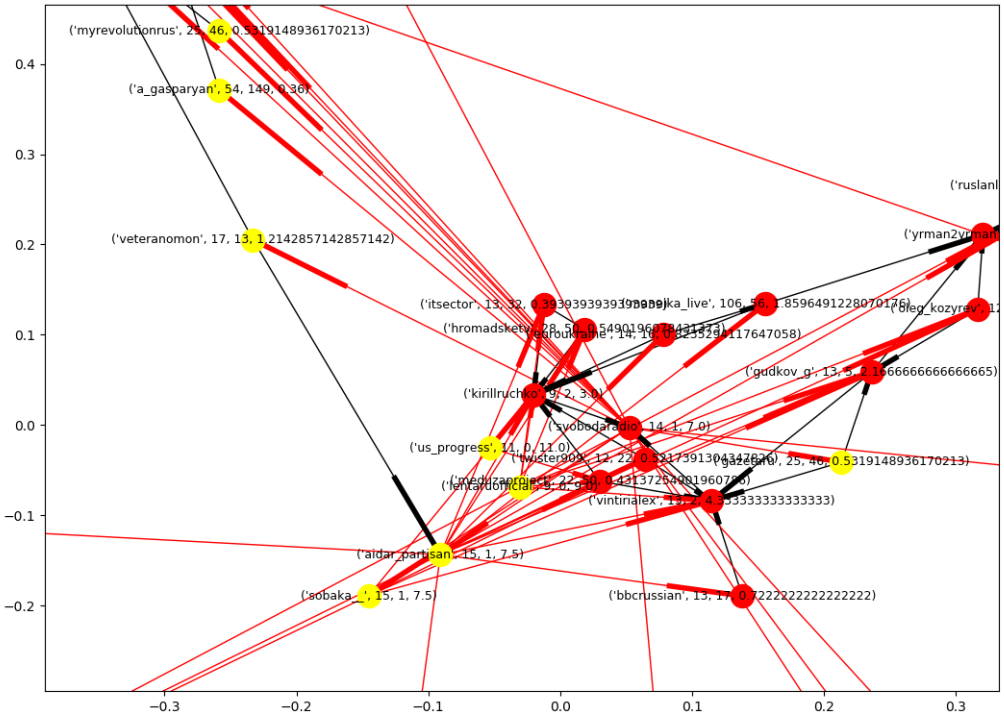


Figura 4.6: Porzione dell'*output* del *tool* a seguito dell'esecuzione di *non greedy* sul *retweet graph #russia_march*.

Capitolo 5

Conclusioni e sviluppi futuri

Bibliografia

[1] J. C. Losada R. M. Benito A. J. Morales J. Borondo. «Measuring Political Polarization: Twitter shows the two sides of Venezuela». In: (2015), pp. 1–10. DOI: 10.1063/1.4913758. URL: https://arxiv.org/pdf/1505.04095.pdf.

[2] Charalampos E. Tsourakakis Cameron Musco Christopher Musco. «Minimizing Polarization and Disagreement in Social Networks». In: (2017), pp. 1–19. URL: https://arxiv.org/pdf/1712.09948.pdf.

[3] C. F. Van Loan G. H. Golub. «Matrix computations». In: (2012).

[4] Aristides Gionis Michael Mathioudakis Kiran Garimella Gianmarco De Francisci Morales. «Reducing Controversy by Connecting Opposing Views». In: (2017), pp. 1–10. DOI: 10.1145/3018661.3018703. URL: https://melmeric.files.wordpress.com/2010/05/reducing-controversy-by-connecting-opposing-views.pdf.

[5] M. E. J. Newman M. Girvan. «Community structure in social and biological networks». In: (2001), pp. 1–6. URL: http://www.pnas.org/content/pnas/99/12/7821.full.pdf.

[6] W.-T. Fu Q. V. Liao. «Beyond the filter bubble: interactive effects of perceived threat and topic involvement on selective exposure to information. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems». In: (2013), 2359–2368.