# Community detection in Tweets

*Professor Gianvito Pio - Gestione e Analisi dei Big Data*

**Petruzzelli Alessandro**

# Introduzione

Con l'avvento dei Social Network, la rappresentazione dei dati attraverso i grafi si è affermata come uno standard per modellare le interazioni tra gli utenti. Sebbene si tratti di una struttura matematica già studiata da Eulero nel XVIII secolo, negli ultimi anni la teoria dei grafi sta facendo notevoli passi avanti. Basti pensare al *Pagerank*, usato da Google per rendere efficiente il sistema di ricerca, o tutti gli algoritmi di routing che ottimizzano l'indirizzamento di pacchetti scegliendo la migliore strategia di percorrenza del grafo.
L'esplosione dei social ha, tuttavia, portato un altro "problema": la mole di dati con cui "fare i conti" è cresciuta quasi in maniera esponenziale come mostrato in figura 1. Questo ha portato a dover anche trovare soluzioni su come poter analizzare questi dati cercando di distribuire il carico su più nodi.

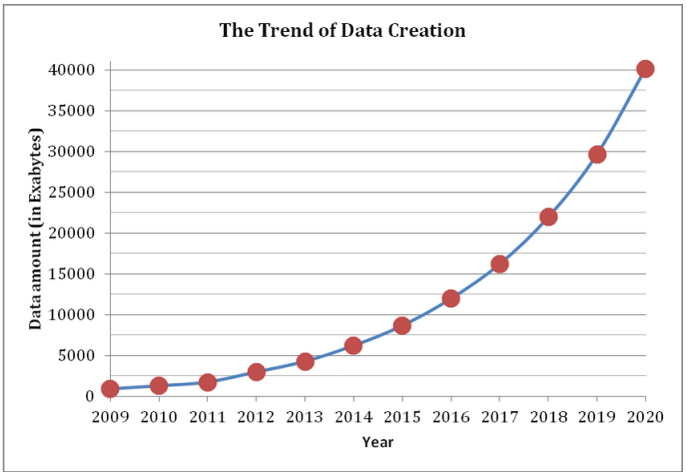


Figura 1: Crescita nella generazione di dati

Per permettere l'analisi e la rappresentazione dei grafi in maniera distribuita, Spark mette a disposizione la libreria GraphX. Più che di una libreria, si tratta di un sistema di processing dei grafi che mette a disposizione strutture dati proprie di spark (come gli RDD) per la manipolazione dei grafi. Inclusi nella libreria ci sono algoritmi base per l'elaborazione dei grafi, come il calcolo del cammino più breve o algoritmi di community detection. All'interno di questo contesto si inserisce il seguente lavoro in cui sono stati implementati tre algoritmi di community detection.

# Stato dell'arte

Gli algoritmi implementati sono stati proposti in un lavoro dal titolo *Uncovering Active Communities from Directed Graphs on Distributed Spark Frameworks, Case Study: Twitter Data*[1]. Gli autori affrontano il problema delle community detection da effettuare su grandi grafi orientati in maniera distribuita. Nel lavoro sono descritti 3 tipi di grafi (o sottografi):

- **Similar interest communities (SIC)**: Un gruppo di persone che rispondono allo stesso evento. Nel caso di Twitter, persone che spesso retwittano o rispondono agli stessi tweets. Questo implica che si condividono gli stessi interessi;
- **Strong-interacting communities (SC)**: Un gruppo di persone che interagiscono tra loro;
- **Strong-interacting communities with their "inner circle" neighbors (SCIC)**: Estensione del SC in cui, al gruppo di persone che interagiscono tra loro, vengono aggiunti membri "esterni" che hanno hanno contattato un membro della community frequentemente.

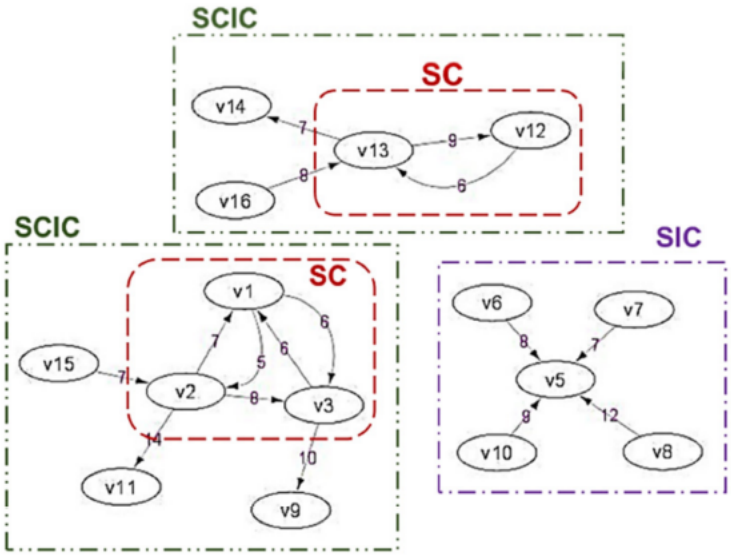


Figura 2: Differenza delle Community

Sebbene figura 2 mostra una netta divisione delle community i 3 algoritmi non forniscono, per noto, un unico gruppo di appartenenza: **un nodo può appartenere a più comunità**. Questo, in fase di visualizzazione delle community, può causare confusione in chi osserva la divisione dei gruppi.
Per ogni gruppo definito precedentemente, nel lavoro di riferimento, sono forniti pseudocodici di 3 algoritmi che ne permettono le identificazioni.

**Algoritmo di SIC detection** L'algoritmo per il ritrovamento delle **Similar Interest Communities** è riportato in figura 3. Il codice implementato presenta alcune differenze; in particolare non sono stati utilizzati Dataframe ma si è preferito mantenere la struttura di RDD. Inoltre si è evitato il coalesce dei dati in un unico nodo per ottenere il DataFrame finale, preferendo un map con (ID Nodo, ID Community).
Il collect è stato effettuato solo in fase di salvataggio su file dei risultati.

**Algorithm 1**: DetectSIC

Descriptions: Detecting SIC from a directed graph using GraphFrame
Input: Directed graph *G*, *thWC1* = threshold of *w*; *thIndeg* = threshold of vertices in-degree
Output: Communities stored in map structure, *comSIC* = map(*CenterId*, list of member *Ids*); a vertex can be member of more than one community.
Steps:
(1) Graph preparation: (a) *filteredE* = *E* in *G* with $w > thWC1$ //Only edges having $w > thWC1$ is used to construct the graph; (b) *GFil* = (*V*, *filteredE*)
(2) Compute *inDeg* for every vertex in *GFil*, store into dataframe, *inD*(*Id*, *inDegree*)
(3) selInD (IdF, inDegree) = inD where inDeg > thIndeg
(4) Find communities: (a) *dfCom* = (*GFil* where its nodes having *Id* = *selInD.IdF*) inner join with *E* on *Id* = *dstId*, order by Id; (b) Collect partitions of *dfCom* from every worker (coalesce) then iterate on each record: read Id and *srcId* column, map the pair value of (Id, *srcId*) into *comSIC*

Figura 3: Algoritmo SIC

**Algoritmo di SI detection** L'algoritmo di **Strong-Interacting Communities** è, in realtà, basato su un algoritmo già implementato nella libreria Graphx. Si tratta del metodo **Strongly Connected Components (SCC)**. All'interno di un grafo questo algoritmo trova i sottografi che risultano completi[1]. Questo algoritmo, applicato ad un grafo connesso, porta al ritrovamento dei cicli tra i nodi.
Anche in questo caso l'algoritmo non è stato implementato in maniera pedissequa. In questo caso la grande differenza è data dall'implementazione del SCC (che produce un RDD con le coppie (ID Nodo, ID Community). Si rende inutile il coalence dei dati e la parte *findComSC*.

**Algorithm 4**: DetectSC-with-SCC

Descriptions: Detecting SC using SCC algorithm
Input: Directed graph *G*; *thWC1* = threshold of *w*; *thDeg* = threshold of vertices degree; *thCtMember* = threshold of member counts in an SCC
Output: Communities stored in map structure, *comSIC* = map(*CenterId*, list of member *Ids*). A vertex can be member of more than one community.
Steps:
(1) Graph preparation: (a) *filteredE* = *E* in *G* with *w* > *thWC1*; (b) *GFil* = (*V*, *filteredE*)
(2) *filteredG* = subgraph of *GFil* where each vertex has degree > *thDeg*
(3) Find strongly connected components from *filteredG*, store as *sccs* dataframe
(4) Using group by, create dataframe *dfCt*(*idCom*,*count*), then filter with *count* > *thCtMember* // The SCC algorithm record every vertex as a member of an SCC (SCC may contain one vertex only)
(5) Join *sccs* and *dfCt* on *Id* = *IdCom* store into *sccsSel* // *sccsSel* contains only records in a community having more than one member
(6) Collect partitions of *sccsSel* from every worker, sort the records by *idCom* in ascending order, then call *findComSC*(*sccsSel*)

**Algorithm 5**: *findComSC*

Descriptions: Formatting communities from *sccsSel*
Input: Dataframe containing connected vertices, *sccs*(*id*, *idCom*)
Output: Communities stored in map structure, *comSCC*: map(*idCom*: String, *ids_count*: String).
*idCom*: string of community Id, *Ids_count*: strings of list of vertex *Ids* in a community separated by space and count of *Ids*.
Steps:
(1) *prevIdCom* = ""; *keyStr* = ""; *addStatus* = false; nMember = 0
(2) For each *row* in *sccsSel*
(3) *line* = *row*; parse *line*; *strId* = line[0]; *idC* = line[1]
(4) if line is the first line: add strId to keyStr; prevIdCom = idC;
nMember = nMember + 1
(5) else if *prevIdCom* == *idC* and *line* is not the last line: add *strId* to *keyStr*;
*nMember* = *nMember* + 1
(6) else if *prevIdCom* != *idC* and *line* is not the last line: add *strId* to *keyStr*,
*nMember* = *nMember* + 1; add (keyStr, nMember) to comSCC;
keyStr = ""; nMember = 0; addStatus = true;
//Final check
(7) if *addStatus* == false and *line* is the last line: add (*keyStr*, *nMember*) to *comSCC*; *addStatus* = true;

Figura 4: Algoritmo SC

[1]comunque prendo due vertici c'è un arco che li unisce

**Algoritmo di SCIC detection** L'algoritmo di **Strong-interacting communities with their "inner circle" neighbors** parte dai rusiltati di alcuni step dell'algoritmo precedente espandendo la community a nodi "esterni". Considerando le differenze già descritte per l'algoritmo SC; questo codice è quello rispettato quasi del tutto. Le differenze principali sono principalmente dalla struttura dei dati utilizzata.

**Algorithm 6**: DetectSCIC-with-SCC

Descriptions: Detecting SCIC using SCC algorithm
Input: Directed graph *G*; *thWC1* = threshold of *w*; *thDeg* = threshold of vertices degree; *thCtMember* = threshold of member counts in an SCC
Output: Communities stored in map structure, *comSCIC* = map(*idCom*: String, *ids_count*: String). A vertex can be member of more than one community.
Steps:
Step 1 to 6 is the same with the ones in *DetectSC-with-SCC*.
(7) Join *sccsSel* with *filteredE* based on *sccsSel.id* = *filteredE.dst* store as *dfIntoSCC* with *src* column renamed as *friendId* //neighbor nodes contact SCC nodes
(8) Join *sccsSel* with *filteredE* based on *sccsSel.id* = *filteredE.src* store as *dfFromSCC* with *dst* column renamed as *friendId* // neighbor nodes contacted by SCC nodes
(9) Merge *dfIntoSCC* and *dfFromSCC* into *dfComExpand* dataframe using union operation
(10) Collect partitions of *dfComExpand* from every worker, sort the records by *IdCom* in ascending order, then call *findComSCIC(dfComExpand)* // The schema is: *dfComExpand(id, idCom, friendId)*

**Algorithm 7**: *findComSCIC*

Descriptions: Formatting communities from *dfComExpand*
Input: Dataframe containing connected vertices, *dfComExpand (id, idCom, friendId )*
Output: Communities stored in map structure, *comSCIC*: map(*idCom*: String, *ids_count*: String). *idCom*: string of community *Id*, *Ids_count*: strings of vertex *Id*s in a community separated by space and count of *Id*s the community.
Steps:
(1) *prevIdCom* = ""; *keyStr* = ""; *addStatus* = false; nMember = 0
(2) For each *row* in *dfComExpand*
(3) *line* = *row*; parse *line*; *strId* = line[0]; *idC* = line[1]; *idFr* = line[2]
(4) if *line* is the first line: add *strId* and *idFr* to *keyStr*; *prevIdCom* = *idC*; *nMember* = *nMember* + 2
(5) else if *prevIdCom* == *idC* and *line* is not the last line: add *idFr* to *keyStr*; *nMember* = *nMember* + 1
(6) else if *prevIdCom* != *idC* and *line* is not the last line: add *idFr* to *keyStr*, *nMember* = *nMember* + 1; add (keyStr, nMember) to comSCC; keyStr =""; nMember = 0; addStatus = true;
//Final check
(7) if *addStatus* == false and *line* is the last line: add (*keyStr*, *nMember*) to *comSCC*; *addStatus* = true;

Figura 5: Algoritmo SCIC

# Dataset

Il dataset utilizzato è relativo a interazioni tra utenti su Twitter. Il dataset è ottenuto dal sito della Stanford Univesity ed è spesso utilizzato in lavori che presentano algoritmi di community detection. Il file del dataset utilizzato è `twitter_combined.txt` contenente tutti gli archi della rete. La rete è composta da:

- 81306 nodi;
- 1768149 archi.

Poiché gli algoritmi lavorano su un grafo con archi pesati, si è deciso di utilizzare il numero di occorrenze di quell'arco come peso. Il grafo, nella sua totalità, si presenta come mostrato in figura 6

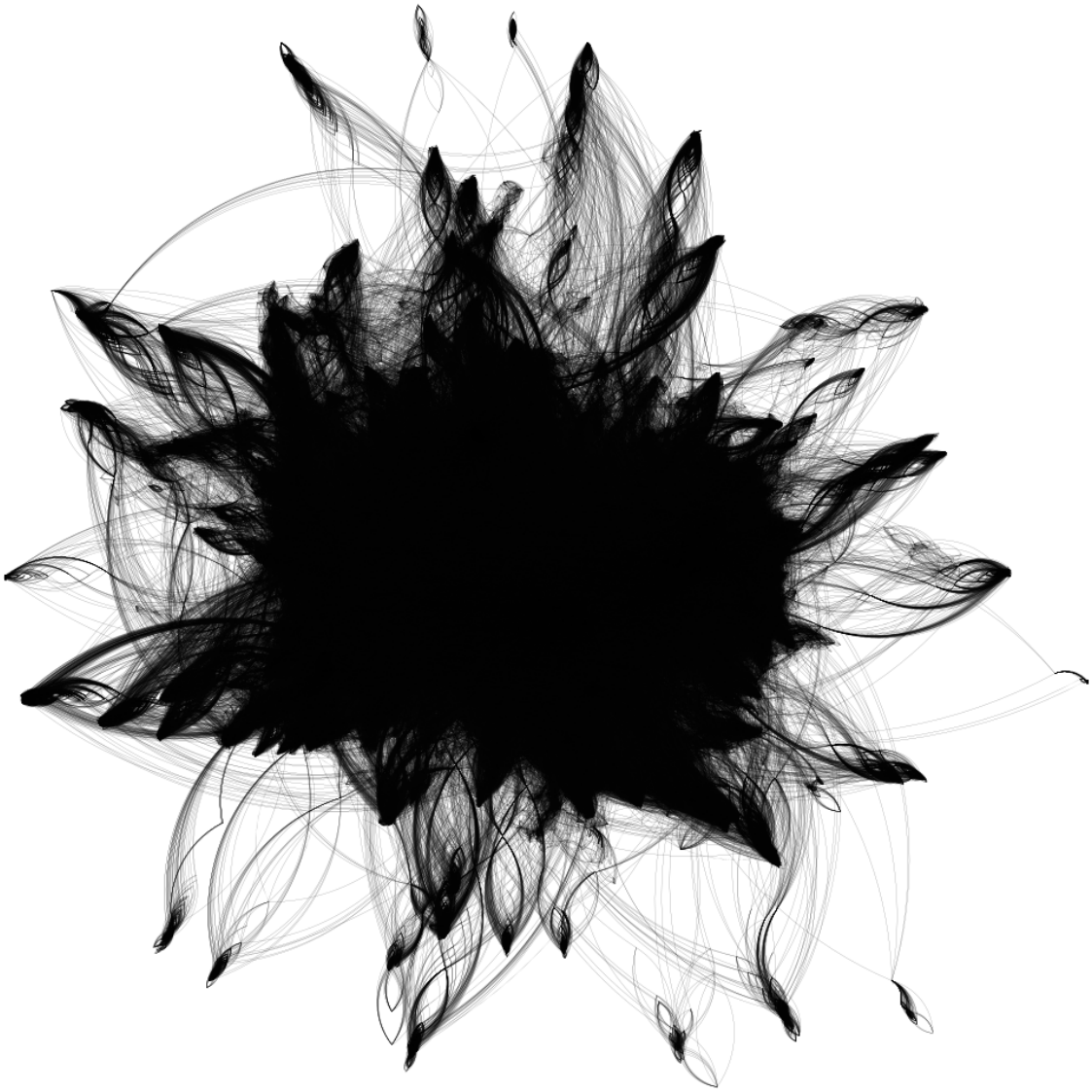

Figura 6: Grafo dell'intera rete

# Risultati

Per l'esecuzione degli algoritmi sono stati selezionati i parametri previsti, i quali sono stati settati a:

- Soglia peso degli archi 5;
- Soglia peso del grado in entrata 3;
- Numero minimo di elementi nella community 5;

I risultati, salvati nella cartella `results`, sono organizzati nel seguente modo:

- `outputGraph<SiglaAlgoritmo>`: Contiene il file txt in cui sono memorizzati gli archi che compongono il grafo sui è eseguito l'algoritmo di community detection (dopo i vari filter e map)
- `outputNode<SiglaAlgoritmo>`: Contiene il file txt in cui sono memorizzati i nodi del grafo. Si tratta di coppie, il primo è l'ID del nodo, il secondo è l'ID del nodo "scelto" come rappresentante della community.

All'interno di ogni cartella è anche presente un file CSV ottenuto dai file precedenti che è stato utilizzato come import al programma Gephi per la rappresentazione.

**SIC** L'applicazione dell'algoritmo SIC sui dati ha prodotto il grafo riportato in figura 7. Si evidenziano 8 community che rappresentano circa il $21,36\%$ di tutti i nodi ovvero circa 410 nodi su 1918 nodi nel grafo, gli altri si possono considerare outlier ad ogni community.

**SC** L'applicazione dell'algoritmo SC sui dati ha prodotto 5 community che coprono il 100% dei nodi. Come di può notare dalla figura 8, c'è una grande predominanza di una community che copre più del 70% dei nodi.

**SCIC** Come prevedibile, l'algoritmo SCIC riduce il numero di cluster rispetto all'SC. Si passa a 4 cluster. In particolare è il secondo cluster (il verde) a vedere incrementati i nodi di "appartenenza" passando da comparire il 16% dei nodi al poco più del 20%.

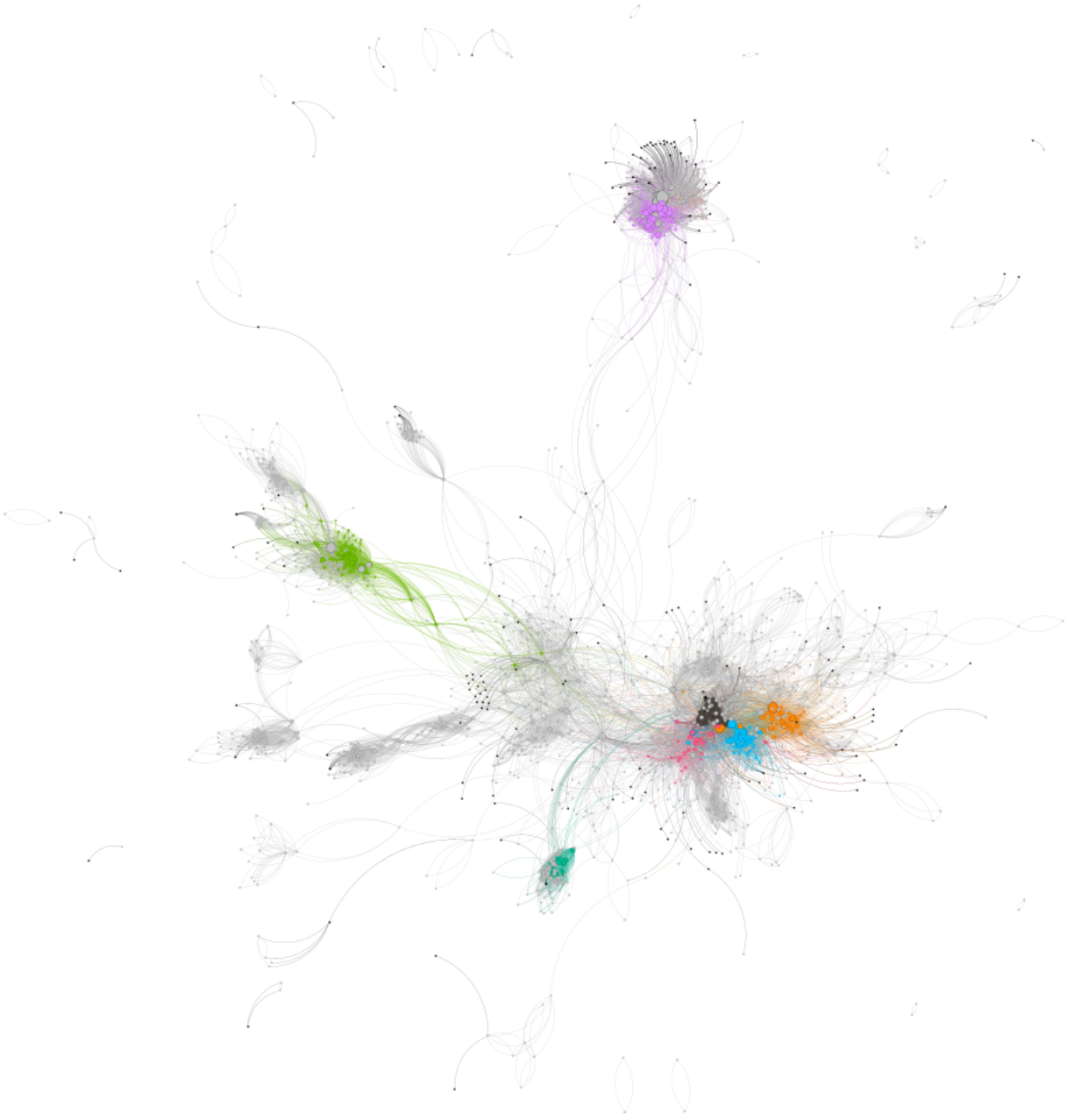

Figura 7: Grafo dopo SIC

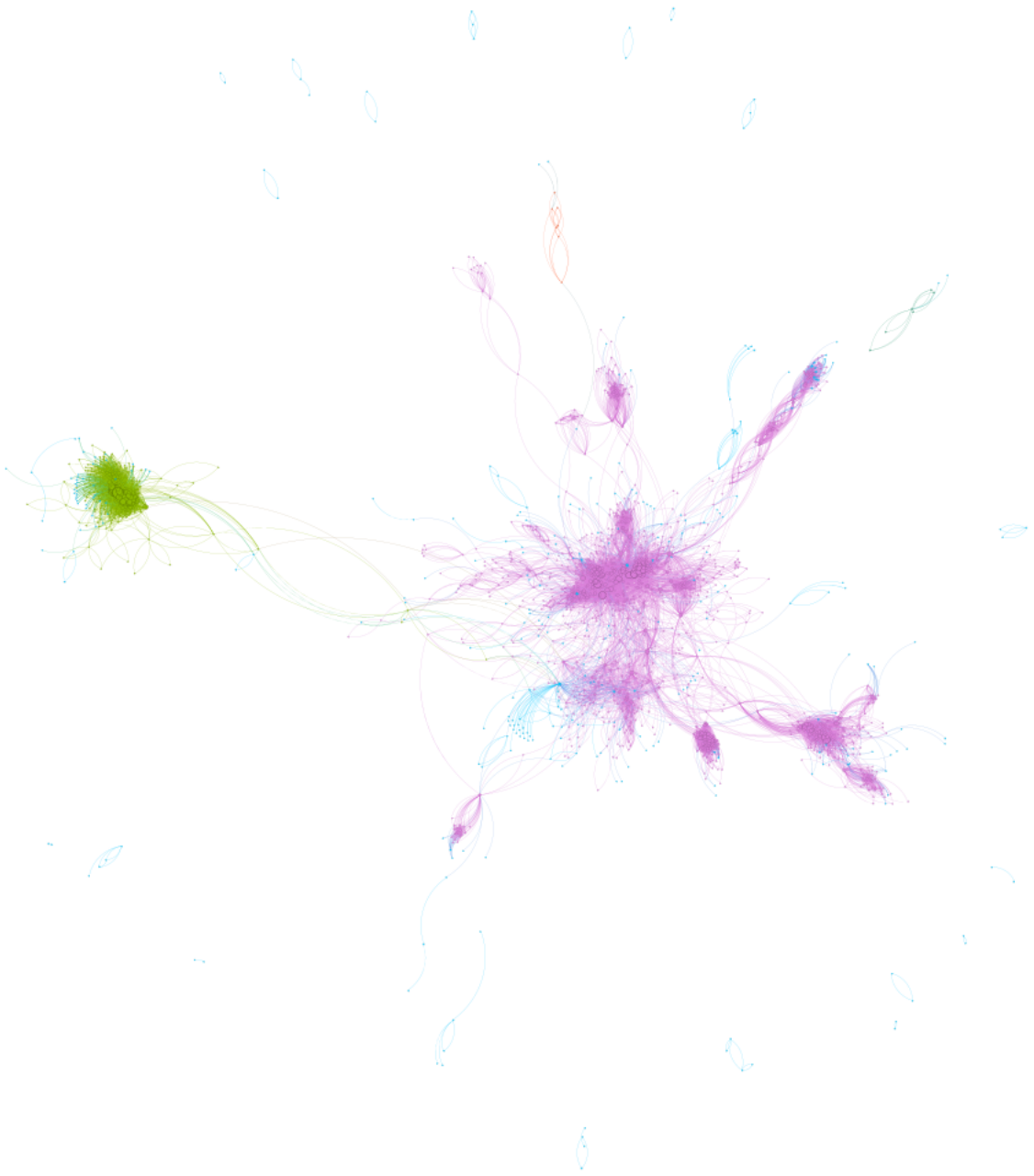

Figura 8: Grafo dopo SC

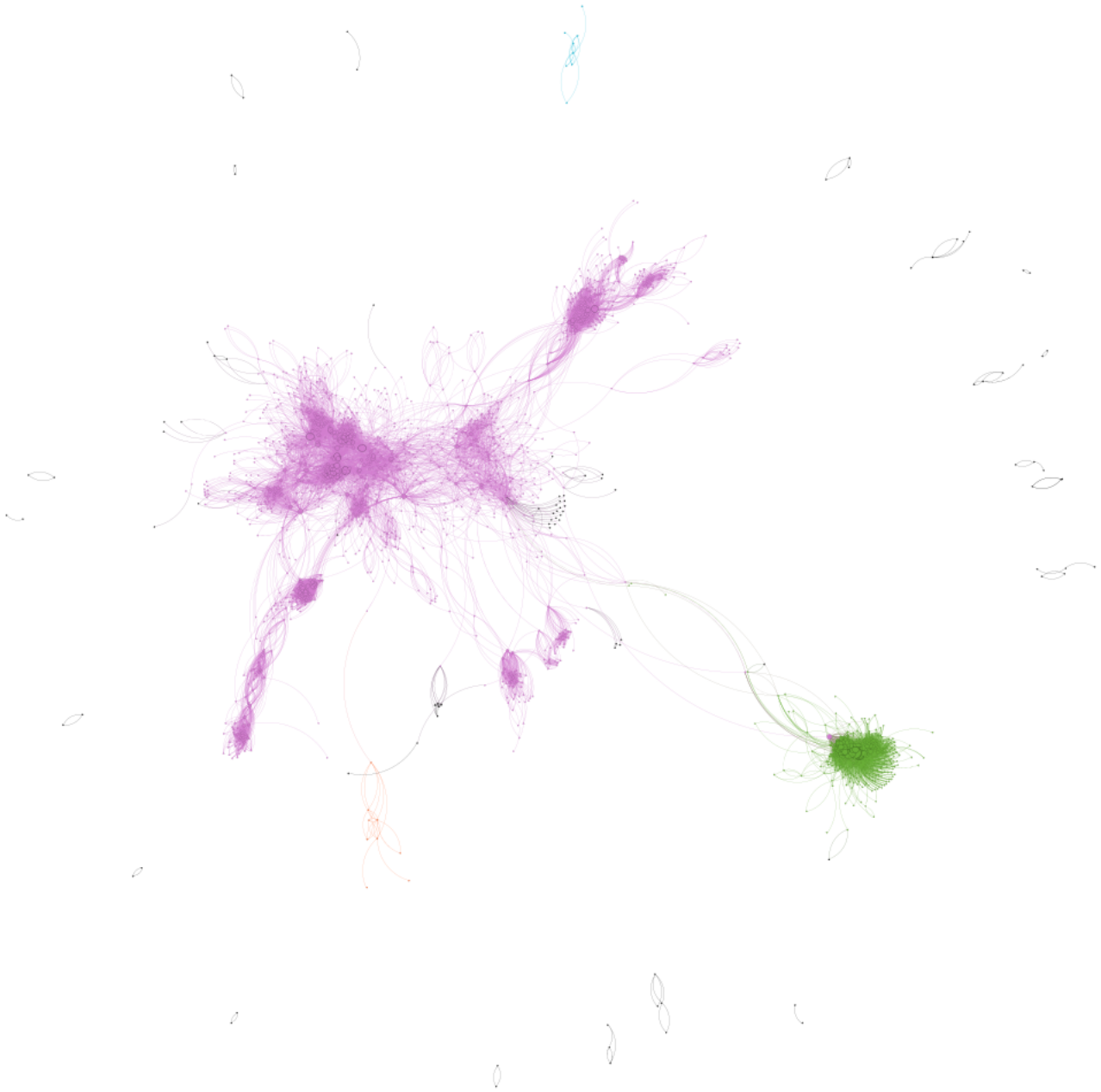

Figura 9: Grafo dopo SCIC

## Riferimenti bibliografici

[1] Veronica S Moertini and Mariskha T Adithia. Uncovering active communities from directed graphs on distributed spark frameworks, case study: Twitter data. *Big Data and Cognitive Computing*, 5(4):46, 2021.

## Note

Il codice è consultabile a repository Github. All'interno, nella cartalla data, oltre al dataset descritto ci sono altri due dataset di tweets. Il primo, relativo al covid è stato scartato perché le community erano poco chiare in quanto si tratta di tweet di account No-Vax che taggano alcuni profili noti come quello delle cause farmaceutiche.
Il secondo è relativo a profili che, nel corso del tempo, hanno parlato di ISIS su twitter. In questo caso i risultati sono più netti, tuttavia si tratta di pochi nodi e interazioni rispetto al dataset preso in esame.
In entrambi i casi, gli algoritmi si possono provare anche su questi file cambiando, all'interno dell'object `CommunityInTweets.scala` la variabile `data_file_path` e `file_tweets`
Tra gli algoritmi implementati c'è anche DENCAST che però non è riportato tra i risultati perché, data la memoria limitata della macchina, l'algoritmo non è mai arrivato a convergenza.