2021

Algoritmo di Louvain

IMPLEMENTAZIONE ED ANALISI DELL'ALGORITMO DI LOUVAIN PER LA COMMUNITY DETECTION

FILIPPO BRAGATO

Sommario

Analisi dell'implementazione di Louvain

Analisi dell'implementazione in Java

Il codice sorgente è disponibile su GitHub a javaLouvain.

Analisi della correttezza

Generazione del grafo

I grafi che sono stati analizzati sono tratti da http://cs.joensuu.fi/sipu/datasets/

In questa dispensa i grafi sono rappresentati come insiemi di punti in un piano cartesiano. Per renderli coerenti con un grafo è necessario collegare tra loro coppie di punti, la scelta di quali punti collegare e di quanto far pesare l'arco che li unisce influenza molto l'output dell'algoritmo.

Nella implementazione proposta sono presenti tre possibili soluzioni di questo primo problema tutte hanno una complessità asintotica non migliore di $O(n^2)$. In ognuna di queste soluzioni il peso dell'arco è definito come il reciproco del quadrato della distanza dei due nodi che l'arco collega.

- graphFromTxtFixedDegree: Unisce ogni nodo *v* ad un numero fissato di nodi *k* scegliendo questi nodi in modo che siano i *k* più vicini a *v*.
- graphFromTxtFixedDistance: Unisce ogni nodo *v* a tutti i nodi w_i che distano meno di una distanza *d* fissata da *v*, in questo modo il grado di ogni nodo è variabile.
- graphFromTxtFull: Unisce tutte le possibili coppie di nodi distini all'interno del grafo.

Si nota che la decisione di imporre il peso di ogni arco come il reciproco del quadrato della distanza tra i nodi che l'arco collega presenta delle criticità.

Infatti, come si può notare dal calcolo di pathbased, il fatto che ci sia una coppia di nodi con coordinate uguali [26.4; 14.35] implica che l'arco che li unisce abbia un peso infinito e che quindi la modularità del grafo non appartenga ai reali. Inoltre in questo modo il valore del peso degli archi differisce molto per istanze simili le cui coordinate di tutti i punti sono semplicemente moltiplicate per una costante.

Per questo vengono aggiunti i seguenti metodi di creazione, hanno un comportamento simile a graphFromTxtFull, infatti collegano ogni possibile coppia di vertici distinti all'interno del grafo, inoltre:

- graphFromTxtSafe elimina nodi doppi con le medesime coordinate.
- graphFromTxtScaled con un fattore di proporzionalità fa in modo che ogni arco abbia un peso sempre non maggiore di 1024.
- graphFromTxtSafeScaled procede con entrambe le operazioni precedentemente descritte.

Ordine dei nodi

Si nota che l'ordine con cui vengono scanditi i nodi ad ogni iterazione dell'algoritmo influenza notevolmente il risultato ottenuto.

Al fine di verificare se fosse possibile trovare un ordinamento univoco con cui ottenere risultati migliori sono state scritte varie implementazioni della classe Comparator.

Non trovando un criterio generale con cui ordinare i nodi in modo che per istanze diverse il risultato sia più vicino a quello sperato, dopo aver preso in analisi qualche possibile ordinamento spaziale e un ordinamento sul grado pesato dei nodi, ho deciso di ridistribuire i nodi in ordine casuale ad ogni iterazione.

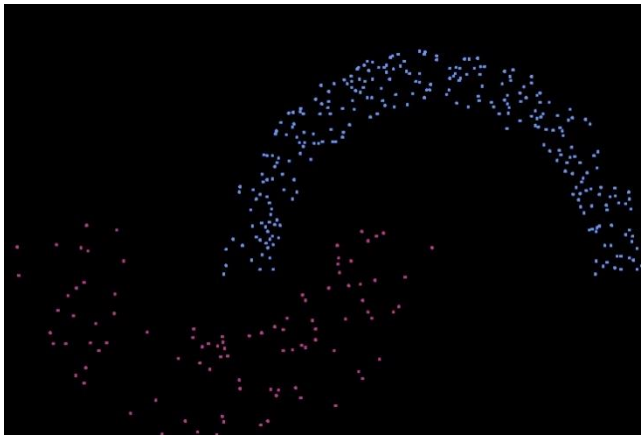
Ripetendo più volte il calcolo dell'algoritmo, poi, viene scelto come output il grafo che presenta la modularità più alta.

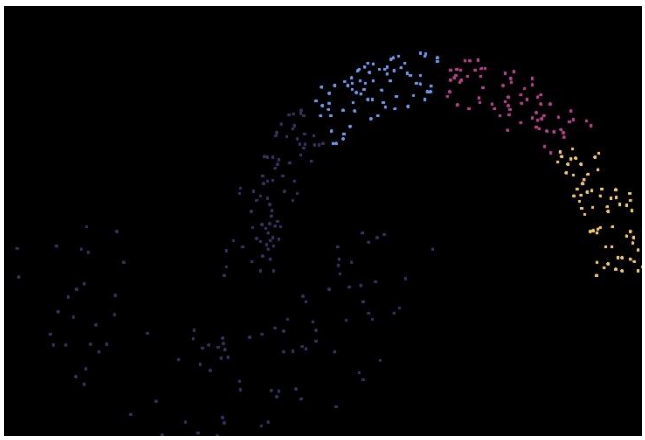
Communities di piccole dimensioni

Si nota che l'algoritmo tende a considerare piccole communities come parte di una sola community, provando a riapplicare l'algoritmo con IterLouvain.java si nota che non è possibile migliorare la modularità ulteriormente.

Communities a bassa modularità, gli shape set

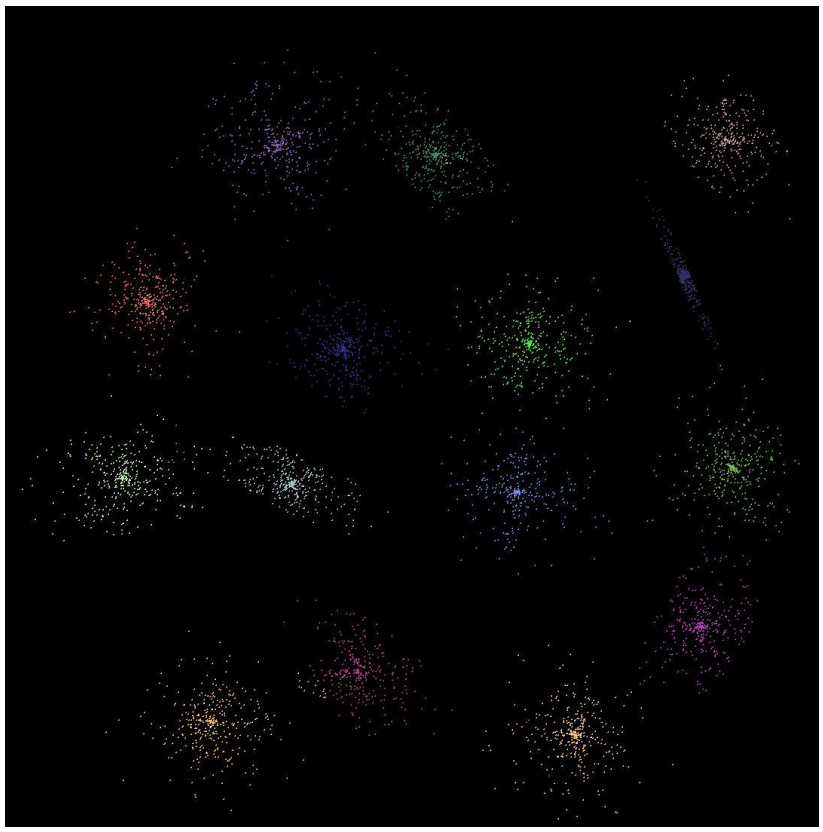
Per quanto riguarda gli shape set, in particolare Jain si nota come la soluzione proposta dal sito dell'università finlandese (nell'immagine a sinistra), per quanto possa sembrare ragionevole, abbia una modularità molto più bassa (0,145) rispetto a quella calcolata dall'output di quest'implementazione dell'algoritmo di Louvain (0,909, nell'immagine a destra), si suppone quindi che questo algoritmo non sia in grado di trovare con precisione delle communities che abbiano caratteristiche analoghe a quelle degli shape set.

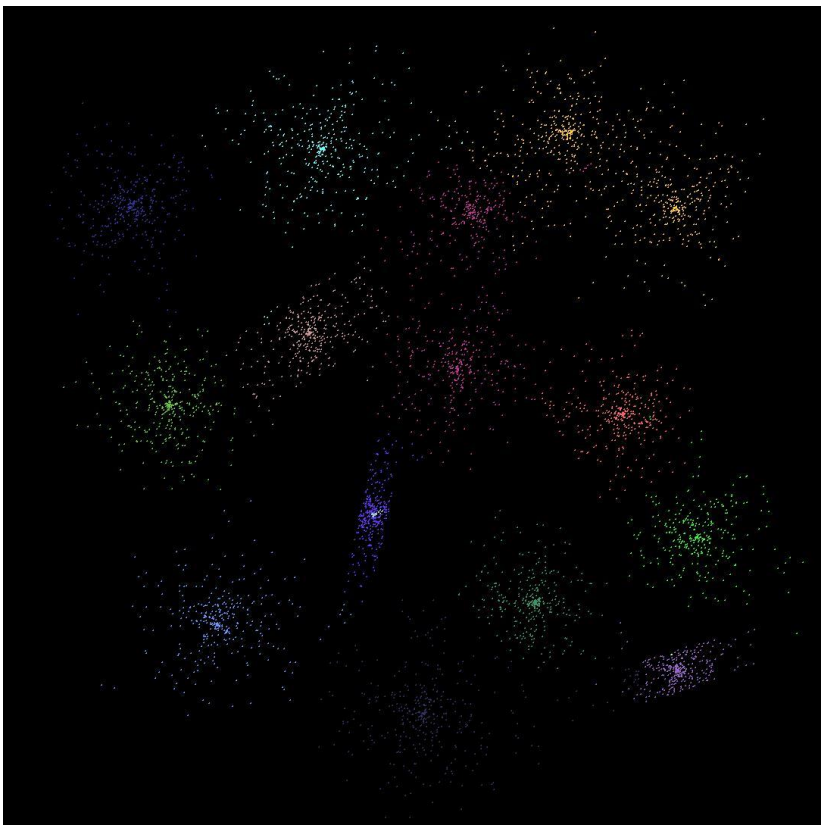




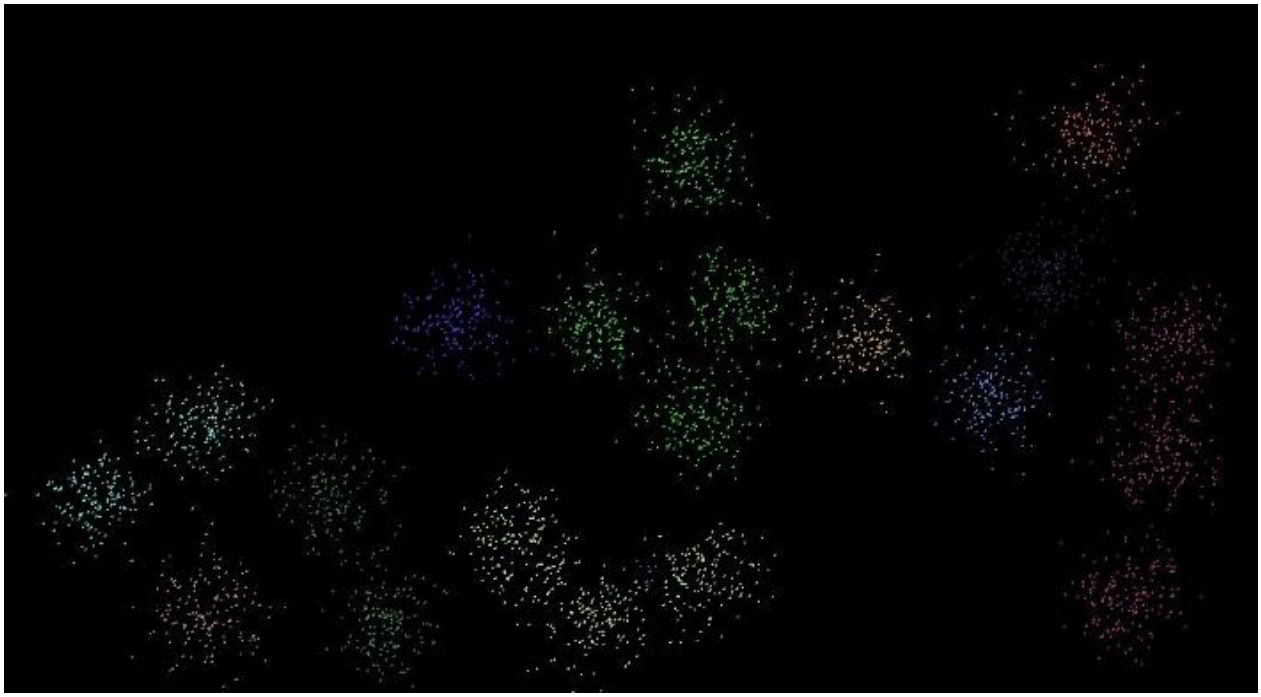
Analisi degli output

L'algoritmo è in grado di riconoscere facilmente gli S-set, ossia insiemi di punti a distribuzione gaussiana attorno a vari centri, anche costruendo il grafo con graphFromTxtFull ossia collegando ogni possibile coppia di punti

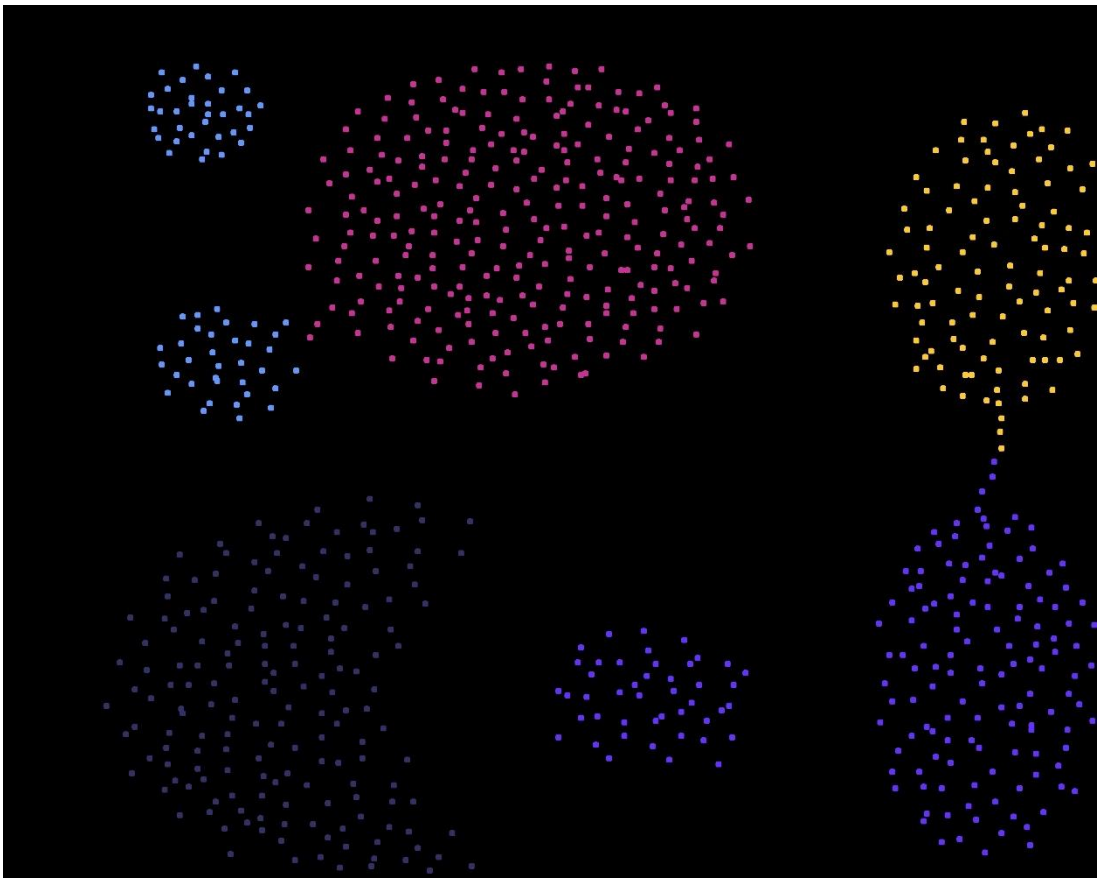


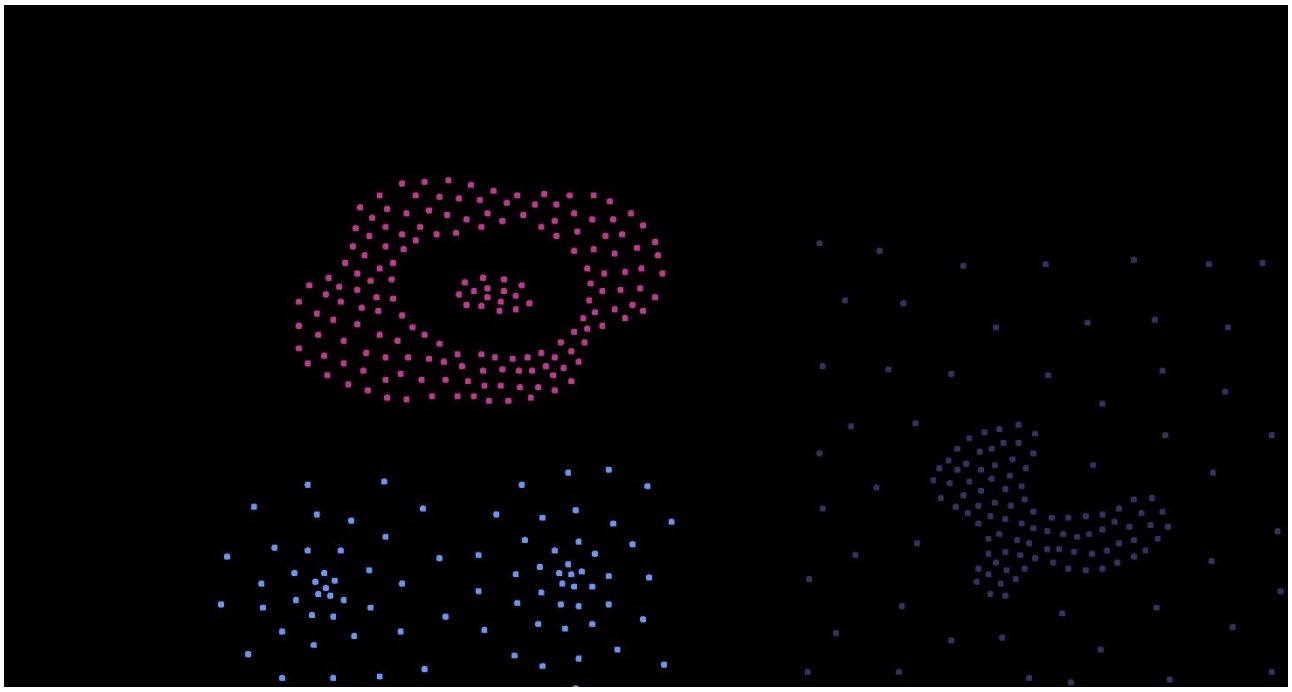


Quando però le communities sono più piccole l'algoritmo non è in grado di distinguerle



Questo si nota molto negli shape set.





Analisi della complessità asintotica dell'algoritmo.

Siano:

- n il numero di nodi,
- m il numero totale degli archi,
- m_i il numero degli archi sul nodo in considerazione,
- c il numero di communities,
- g la loro grandezza media;

In ordine vengono eseguiti i seguenti cicli:

- O(n): il ciclo che scandisce tutti i nodi
- O($n \cdot m_i$): il ciclo che inserisce nei nodi il loro grado pesato
- O(m): il ciclo che trova il totale del peso degli archi
- O(n): il while principale (infatti ad ogni iterazione di questo ciclo viene tolto almeno un nodo dal grafo, se il nodo non viene tolto l'algoritmo è arrivato alla fine)
    - O(n): l'attraversamento di tutti i nodi
    - O(m_i): il ciclo per capire se ho un self loop
    - O(m_i): il ciclo per trovare tutti i nodi vicini al nodo in esame
    - O(m_i): il ciclo per calcolare la variazione di modularità
    - O(m_i): il ciclo per trovare il massimo di tale variazione
    - O(m_i): il ciclo per salvare il peso degli archi
    - O(m_i): il ciclo per rimuovere gli archi
    - O(m_i): il ciclo per sistemare gli archi una volta rimosso il nodo

Più nello specifico ad ogni iterazione del ciclo while o il numero di nodi decresce o il ciclo termina, ne consegue il fatto che il ciclo while esegue al più n operazioni. Si nota inoltre che al termine del ciclo i nodi rimanenti rappresentano le community. Quindi n decresce fino a raggiungere c e quindi possiamo ridurre il numero di iterazioni massime ad n-c.

La taglia di n quindi, all'interno del ciclo while si riduce ad ogni iterazione. Ne consegue che il numero di iterazioni nel ciclo che scandisce tutti i nodi all'interno del grafo non è sempre n. Per le stesse motivazioni già espresse n deve diminuire ad ogni ciclo esterno affinché l'algoritmo non finisca la sua esecuzione.

Al caso pessimo

Supponendo il caso pessimo, ossia che ad ogni iterazione il numero di nodi diminuisca solo di uno, il numero di operazioni totali sarà:

$$\sum_{i=0}^{n-c}[(n-i)+7\cdot m_i] = \sum_{i=0}^{n-c}(n-i) + 7\sum_{i=0}^{n-c} m_i = (n-c)\cdot n - \sum_{i=1}^{n-c} i + 7\cdot(n-c)\cdot m_c$$
$$= n^2 - nc - \frac{(n-c)(n-c+1)}{2} + 7nm_c - 7cm_c$$
$$= \frac{2n^2 - 2nc - n^2 + nc - n + nc - c^2 + c + 14nm_c - 14cm_c}{2}$$
$$= \frac{n^2 - n - c^2 + c + 14nm_c - 14cm_c}{2} = O(n^2)$$

Poiché $n > c$ e $n > m_c$.

Al caso medio

Al caso medio, tuttavia, possiamo verosimilmente affermare che non venga eliminato un solo nodo per ogni iterazione del ciclo while, ma un numero proporzionale ad *n*, con costante di proporzionalità $k \in (0,1)$.

Secondo questa ipotesi, al primo ciclo while il for esegue *n* operazioni, al secondo ne esegue *(1-k)n,* all'*i*-esimo ciclo esegue *(1-k)^{i}n* operazioni.

Quindi il ciclo esterno viene iterato un numero τ di volte tale che $(1-k)^\tau n = c$

$$\log_2(1-k)^\tau n = \log_2 c$$
$$\tau \log_2(1-k) + \log_2 n = \log_2 c$$
$$\tau = \frac{\log_2 \frac{c}{n}}{\log_2(1-k)}$$

Il numero totale di operazioni quindi sarà:

$$\sum_{i=0}^{\tau}[(1-k)^i n + 7\cdot m_i] = n\sum_{i=0}^{\tau}(1-k)^i + \tau\cdot 7\cdot m_i = n\frac{1-(1-k)^{\tau+1}}{k} + \tau\cdot 7\cdot m_i = O(n)$$

Analisi del tempo di esecuzione

Con numero di vertici fissato

Si tengono in considerazione gli stessi parametri usati nel caso del calcolo della complessità asintotica. In particolare, si nota che per gli input dati

- *n=5000*
- *m =*

Grado massimo	S1	S2	S3	S4
10	25 537	25 488	25 464	25 443
100	261 270	259 665	254 654	253 883
1000	2 503 017	2 505 775	2 515 793	2 490 954

Distanza massima	S1	S2	S3	S4
1000	1 039	757	697	863
10 000	52 824	37 552	31 481	42 529
100 000	781 881	748 280	847 233	954 393

- *c =*

Grado massimo	S1	S2	S3	S4
10	30	25	28	26
100	16	16	16	14
1000	16	14	13	9

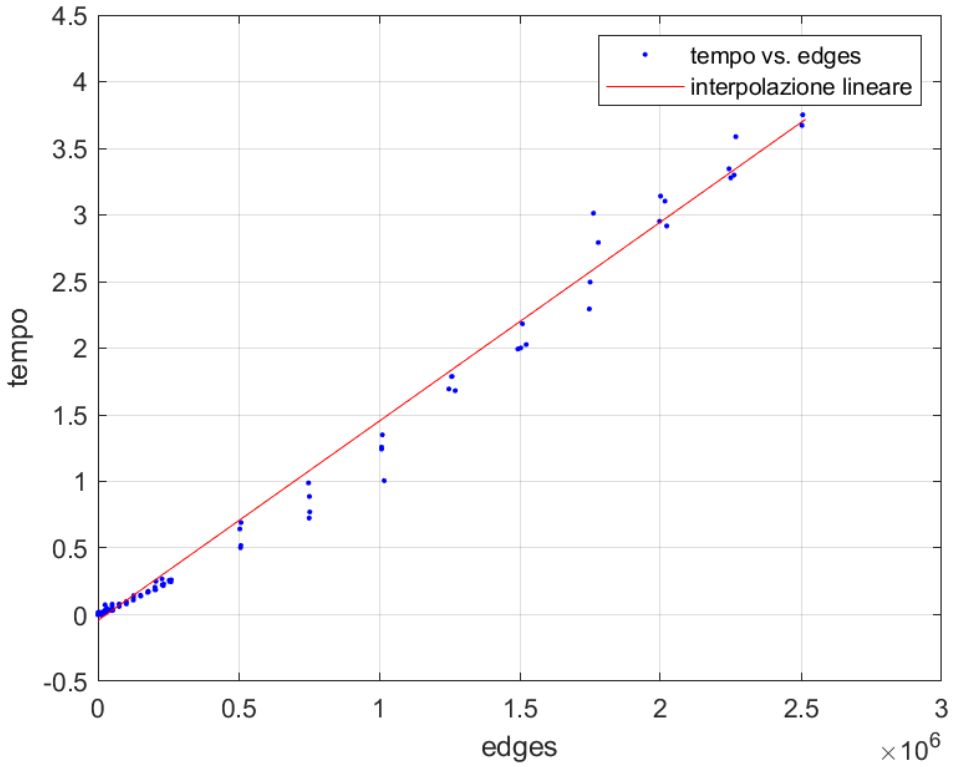
Distanza massima	S1	S2	S3	S4
1000	4 385	4 5525	4 633	4 537
10 000	474	753	764	593
100 000	16	15	10	10

Attraverso dati sperimentali si ricava quanto tempo è necessario all'esecuzione dell'algoritmo.

Grado massimo	S1	S2	S3	S4
10	0.085s	0.055s	0.003s	0.04s
100	0.445s	0.283s	0.027s	0.275s
1000	4.806s	4.649s	5.99s	4.896s

Distanza massima	S1	S2	S3	S4
1000	0.008s	0.003s	0.003s	0.004s
10 000	0.07s	0.03s	0.027s	0.036s
100 000	1.135s	0.698s	0.962s	1.073s

Si ricorda che questo tipo di misure è soggetta ad una varianza molto elevata. Tuttavia è evidente che vi sia una correlazione lineare tra numero di archi e velocità di esecuzione, correlazione che era sfuggita alla precedente analisi asintotica per le approssimazioni troppo grossolane su n.



Da un'analisi più approfondita, prendendo un campione più grande come caso di studio si verifica facilmente la dipendenza lineare del tempo dal numero di archi presenti nel grafo.

Viene riportato un semplice grafico in cui si mettono a confronto il numero di archi e il tempo di esecuzione.

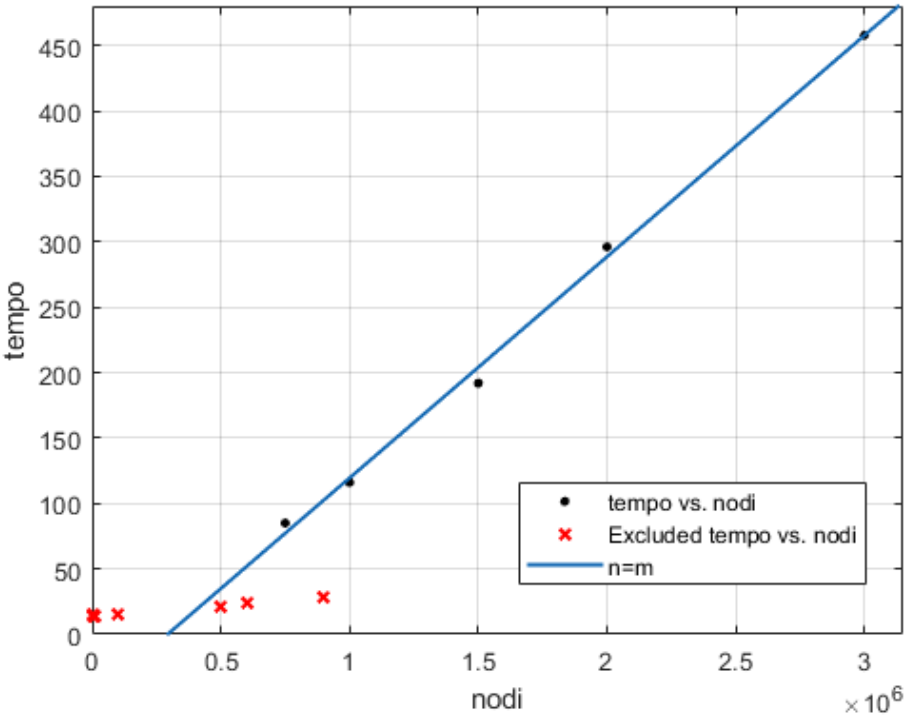
Con numero di archi fissato

Fissando il numero di archi, invece, si ottengono dati molto più interessanti. Per questo studio è stato utilizzato VisualVM, il profiler della Oracle Corporation per programmi Java, per misurare nel modo più preciso possibile il tempo di utilizzo della CPU di quest'implementazione dell'algoritmo di Louvain.

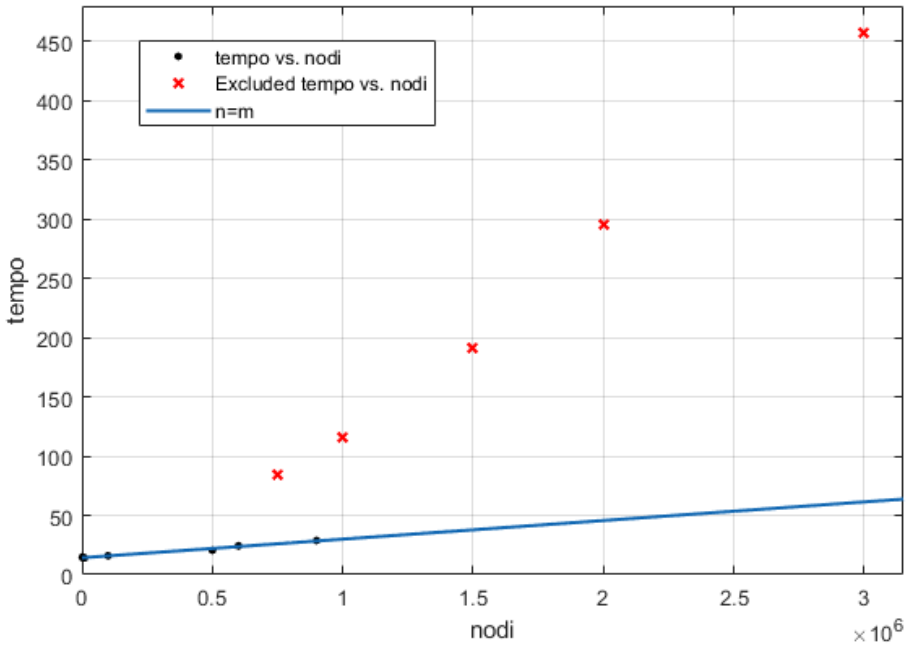
Per ottenere dei grafi con un numero fisso di archi è stato implementato il metodo graphFromTxtFixedArc nella classe GraphGenerator, esso aggiunge un numero fissato di archi *a* tra *a* coppie di nodi scelte in modo casuale. Ed è stata costruita la classe di supporto RandomTxt che si occupa di creare *n* coppie di coordinate casuali intere e di salvarle su un file .txt con un pattern analogo ai grafi tratti dal sito di riferimento.

Per questo studio il numero di archi è stato fissato a 500 000 e si è fatto variare notevolmente il numero di nodi all'interno del grafo.

Si nota che per un numero di nodi confrontabile con il numero di archi del grafo, il tempo di utilizzo della CPU tende a rimanere costante, o meglio, il grafico che mette a confronto il numero di nodi con il tempo è una retta con pendenza $1{,}5 \cdot 10^{-5}$.



Per quanto riguarda, invece, valori di *n* maggiori di 500 000 o addirittura un ordine di grandezza più grande, si nota una dipendenza lineare tra il numero di nodi e il tempo di esecuzione



Conclusioni

Dal punto di vista del calcolo sperimentale si nota quindi come per m>n la complessità temporale dell'algoritmo sia O(m), mentre se n>>m allora la complessità è da considerarsi come O(n).

Analisi dell'implementazione in MatLab

Il codice sorgente è disponibile su GitHub a matlabLouvain.

Per quanto riguarda la correttezza dell'algoritmo si arriva esattamente agli stessi risultati riportati sopra, ben diversa, però è la complessità asintotica e il tempo di esecuzione dell'algoritmo.

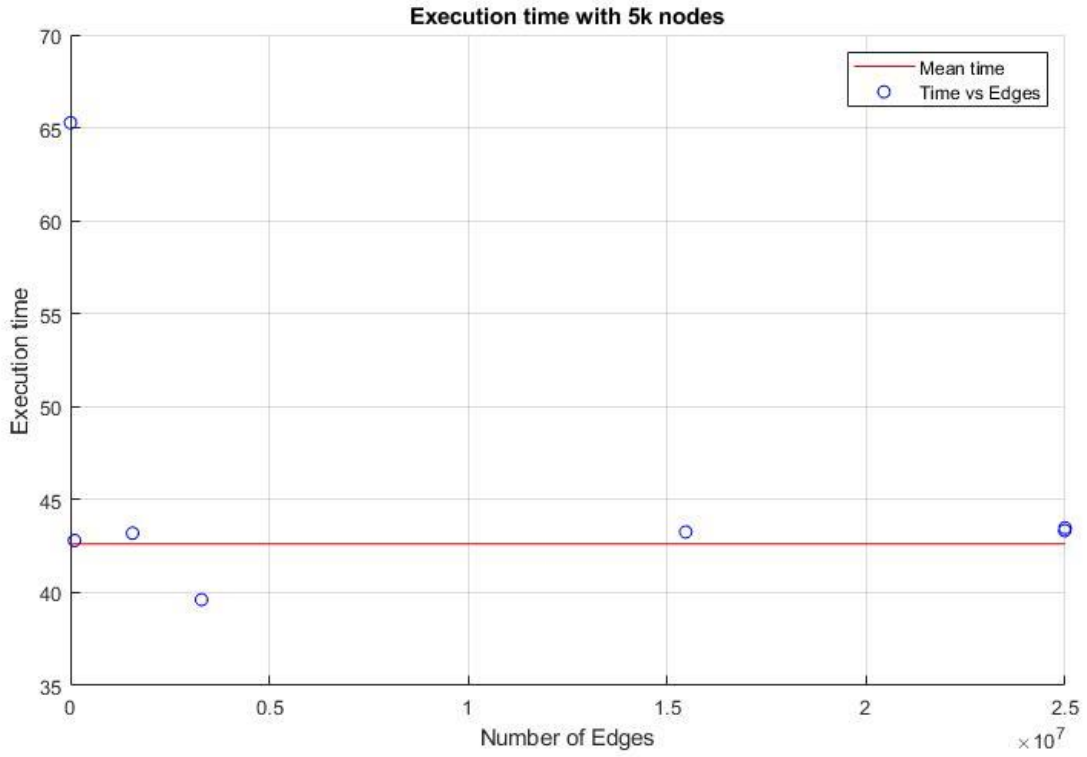
So che MatLab supporta anche una programmazione orientata agli oggetti, ma personalmente ritengo doloroso lavorare con oggetti in un linguaggio in cui non vi è una forte tipizzazione, quindi per implementare l'algoritmo di Lovain ho deciso di rappresentare il grafo con la sua matrice delle adiacenze (che ovviamente ha i suoi pro e contro).

In termini di matrice delle adiacenze, i termini k_i, $k_{i,in}$, Σ_{in} e Σ_{tot} sono particolarmente facili da trovare, in quanto sono rispettivamente la somma della colonna del nodo che sta per cambiare community, la colonna dello stesso nodo, la diagonale della matrice e la somma delle colonne di tutta la matrice.

L'implementazione risulta quindi molto più semplice da scrivere e contemporaneamente molto più difficile da leggere, ma il punto di debolezza più grosso è legato al tempo di esecuzione dell'applicazione.

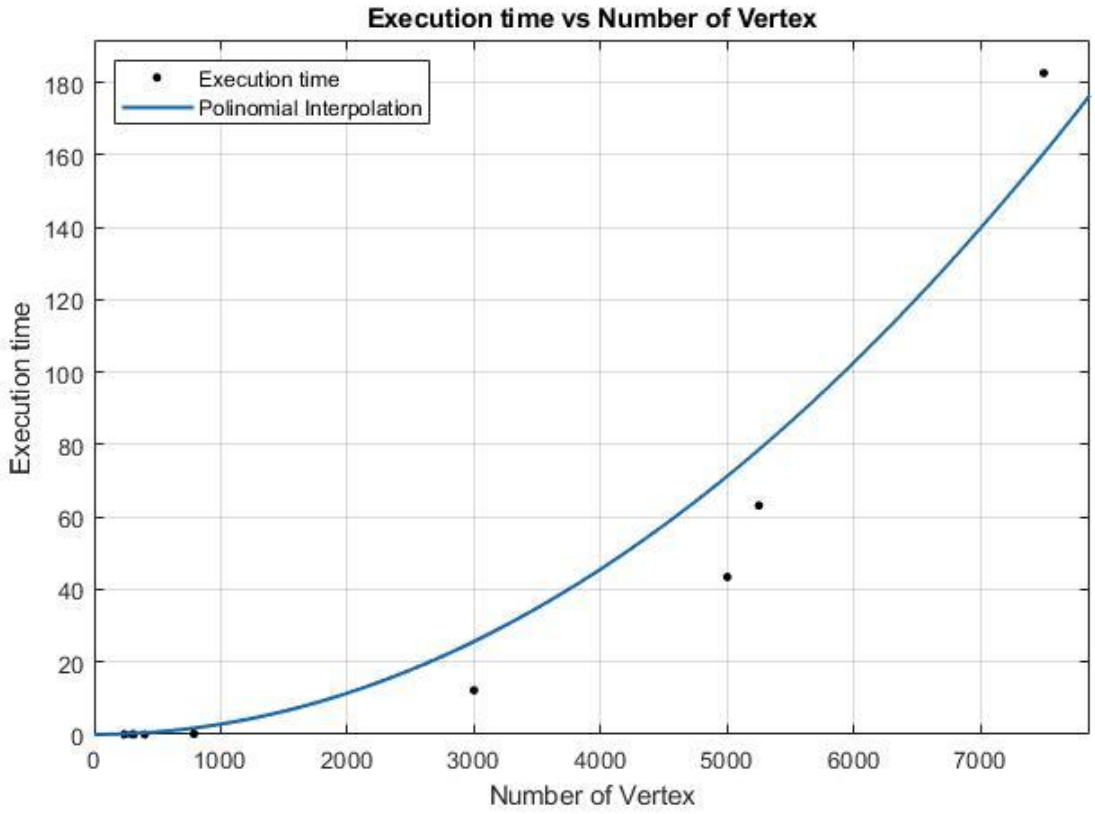
Analisi del tempo di esecuzione

Si nota infatti che il tempo di esecuzione è indipendente dal numero di archi, spiegabile con il fatto che la non esistenza di un arco viene raffigurata come uno zero nella matrice delle adiacenze, quindi come un arco di peso nullo.



Riporto un grafico di alcune misure sperimentali fatte con un grafo di 5 000 nodi e un numero variabile di archi.

Essendo il tempo di esecuzione indipendente dal numero di archi, quindi, si raccolgono dati al variare di nodi ed archi supponendo che il numero di questi ultimi non influenzi il risultato. Ciò che si nota è una dipendenza quadratica del tempo di esecuzione dal numero di nodi, l'implementazione è quindi $O(n^2)$. In sostanza questa implementazione in MatLab si comporta al caso medio come l'implementazione in Java al caso pessimo.



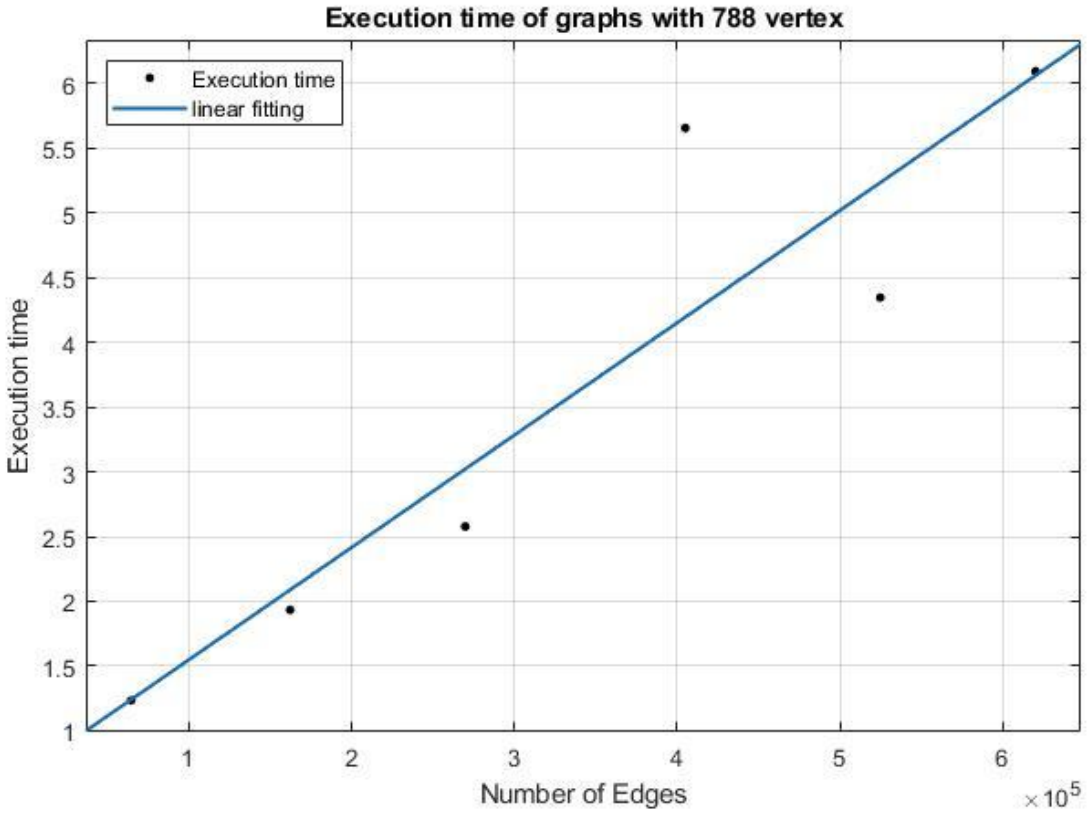
Confronto con implementazioni off the shelf

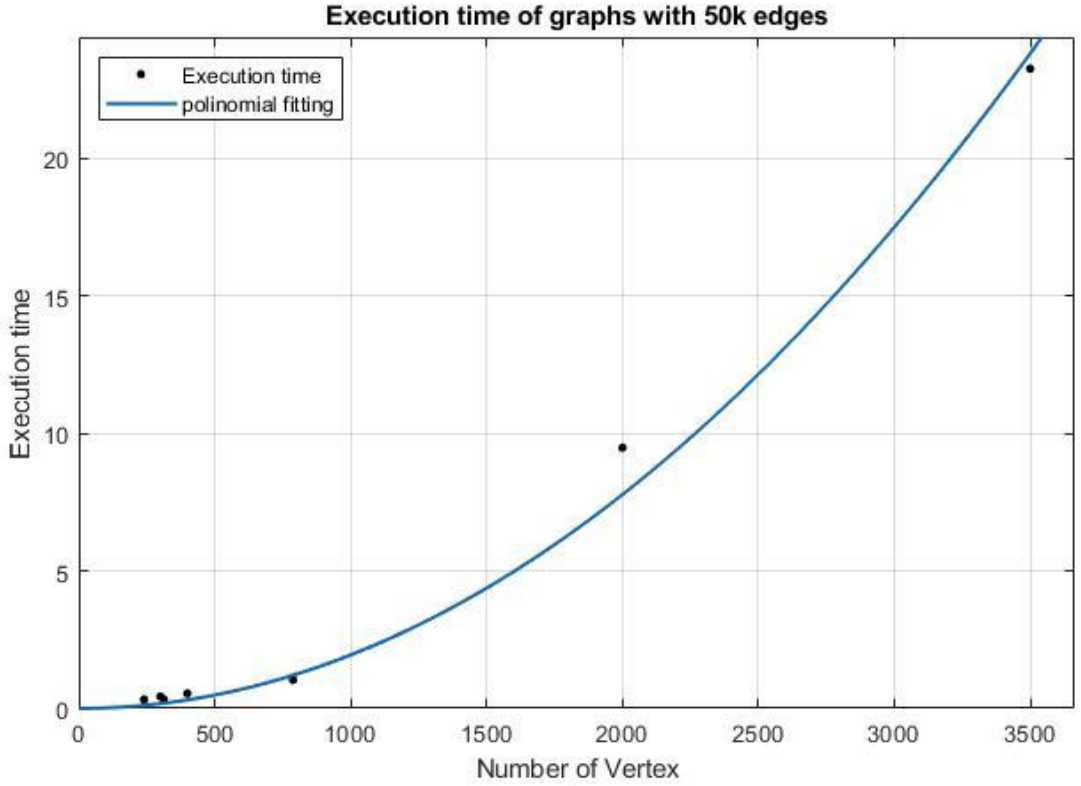
Ho preso in analisi l'implementazione in MatLab di Antoine Scherrer disponibile qui.

Tramite qualche prova ho calcolato in modo empirico la complessità dell'algoritmo, in particolare, fissato il numero di nodi, l'algoritmo ha un comportamento asintotico $O(m)$.

Invece fissando il numero di archi si trova che il tempo di esecuzione è $O(n^2)$.

Seguono i grafici che riportano i dati ottenuti





Conclusioni

Ne consegue che la complessità totale è $O(m \cdot n^2)$. Rispetto alla mia implementazione in MatLab, quindi, potrebbe essere più o meno veloce a seconda di quanto il grafo è densamente connesso.

Fonti

I grafi su cui sono stati testate le implementazioni, riportati nel testo sotto forma di immagini sono tratti da:

P. Fränti and S. Sieranoja
K-means properties on six clustering benchmark datasets
Applied Intelligence, 48 (12), 4743-4759, December 2018
https://doi.org/10.1007/s10489-018-1238-7
BibTex

L'altra implementazione confrontata è opera di:

Antoine Scherrer
https://github.com/epfl-lts2/gspbox/tree/master/3rdparty/Community_BGLL_Matlab
Vedi anche
Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, Etienne Lefebvre
Fast unfolding of community hierarchies in large networks
https://arxiv.org/abs/0803.0476