Algoritmo EAGLE: implementazione e valutazione delle performance

Asara, Dignani, Mandalà, Rafanelli

4 aprile 2011

Studi nel campo del clustering hanno dimostrato che le community sono in genere sia overlapping che gerarchiche; EAGLE (agglomerativE hierarchicAl clusterinG based on maximaL cliquE) è un algoritmo di community detection che considera entrambe le proprietà, a differenza di molti altri. In questo documento mostriamo una possibile implementazione di EAGLE, utilizzando una versione appositamente modificata dell'algoritmo per renderlo più veloce.

Successivamente eseguiamo gli algoritmi EAGLE e GCE (Greedy Clique Expansion) su un set di grafi e ne confrontiamo i risultati, in termini di performance, numero di community rilevate, la loro dimensione varie altre metriche.

1 Introduzione (andrea)

2 L'algoritmo EAGLE (silvia, federico)

2.1 Descrizione algoritmo e caratteristiche (silvia)

2.2 Versione originale (federico)

È possibile implementare l'algoritmo così come proposto dagli autori in questa maniera (in linguaggio Python-Like):

```
1 def EAGLE(k):
2         mc = maximal_cliques(minSize = k)
3         unusedNodes = [nodi che non fanno parte di community]
4
5         for node in unusedNodes:
6                 mc.append([aggiungi una community con solo "node" a mc])
7
8         bestSet = mc
```

```
 9           bestEQ   = EQ(bestSet)
10
11           while mc.size > 1:
12                   a, b = [estrai le communities con similarity più alta]
13                   mc.append([unione tra a e b])
14
15                   actualEQ = EQ(mc)
16                   if actualEQ > bestEQ:
17                           bestSet = mc
18                           bestEQ = actualEQ
19
20           return bestSet
```

Il codice riportato esegue le seguenti operazioni:

1. calcola il set delle communities iniziali:
   a) calcola tutte le clique massimali di dimensione minima k (come già detto, 4 è un buon valore);
   b) ogni nodo del grafo che non fa parte di nessuna community viene visto come una community a se stante;
2. calcola il valore della modularity estesa per il set iniziale, salva il risultato in bestEQ e una copia del set orignale in bestSet;
3. esegue ciclicamente le seguenti operazioni finché non rimane un'unica community nel set:
   a) estrae dal set le communities a e b che assieme danno luogo al più alto valore di similarity;
   b) a e b vengono unite in un'unica community, e questa viene inserita nel set;
   c) se il set ha un EQ più alto di bestEQ questo viene salvato come nuovo bestSet, e bestEQ prende il valore calcolato;
4. bestSet è il set delle communities finale.

Come si può vedere, ci sono tre funzioni a cui si fa riferimento nel listato:

maximal_cliques questa funzione restituisce le clique massimali di un grafo; gli autori di EAGLE suggeriscono l'utilizzo dell'algoritmo Bron-Kerbosch. Non si considerano tutte le clique massimali, ma soltanto quelle che hanno un numero di nodi maggiore o uguale ad una certa soglia, indicata come k nell'articolo. La funzione deve quindi accettare questo parametro.

EQ questa funzione calcola la modularity estesa, così come proposta dagli autori di EAGLE:

$$EQ = \frac{1}{2m} \sum_i \sum_{v,w \in C_i} \frac{1}{O_v O_w} \left[A_{vw} - \frac{k_v k_w}{2m} \right]$$

dove:

- m è il numero di archi nel grafo;
- O_v è il numero di communities a cui il nodo v appartiene;
- k_v è il grado del nodo v;
- A_{vw} è l'elemento w della riga v della matrice di adiacenza del grafo: assume valore 1 se esiste un arco tra i due nodi, altrimenti vale 0.

Un alto valore di EQ indica una struttura di communities con molte sovrapposizioni. EQ vale 0 quando tutti i nodi appartengono alla stessa community, mentre si riduce alla modularità Q quando ogni nodo appartiene ad una sola community.

similarity questa funzione, anche se non appare esplicitamente nel listato, valuta la somiglianza tra due communities, siano C_1 e C_2, ed è così definita:

$$M = \frac{1}{2m} \sum_{v \in C_1, w \in C_2, v \neq w} \left[A_{vw} - \frac{k_v k_w}{2m} \right]$$

ed assume valore tanto più alto quanto più simili sono le due community considerate.

2.3 Modifica dell'algoritmo

Indubbiamente nell'algoritmo così come presentato ci sono diversi *colli di bottiglia*, cioè serie di operazioni molto pesanti e ripetute frequentemente, che rallentano notevolmente l'esecuzione dell'algoritmo. Vediamo ora nel dettaglio quali sono:

2.3.1 Similarity

$$M_{C_1,C_2} = \frac{1}{2m} \sum_{v \in C_1, w \in C_2, v \neq w} \left[A_{vw} - \frac{k_v k_w}{2m} \right] \tag{1}$$

La similarity è la funzione che viene calcolata più volte tra tutte quelle presenti nell'algoritmo. Se si dovesse implementare l'algoritmo così come descritto nel listato precedente si avrebbero tantissime chiamate.

Ad esempio, il grafo IT contiene 611 nodi e 620 communities iniziali; il ciclo while verrà ripetuto 619 volte. Indicando come N il numero attuale di communities, ricordando che

$$620 \geq N \geq 2$$

bisogna calcolare la similarity fra

$$P = \frac{N(N-1)}{2}$$

communities. Allora il numero totale di volte che si calcola la similarity è dato da:

$$\#similarity = \sum_{N=2}^{620} \frac{N(N-1)}{2} = 39721230$$

che è evidentemente un numero molto alto. Il grafo più grande a nostra disposizione ha 35390 nodi e 36496 communities iniziali; i conti di prima indicano quindi che il calcolo della similarity verrà eseguito un numero di volte pari a 8.101.190.658.480, circa $8\,10^{12}$ operazioni. Questa operazione deve essere eseguita velocemente, altrimenti i tempi di attesa saranno molto lunghi.

La funzione similarity, così come riportata sopra non è molto performante, in quanto non consente il precalcolo di alcuni dati. Proponiamo allora una versione modificata della similarity, indicata come $\frac{M_{C_1,C_2}}{f}$, dove:

$$f = \frac{1}{2m} \geq 0$$

viene chiamato fattore di scala. Essendo costante per tutta l'esecuzione di EAGLE (l'algoritmo non modifica le connessioni tra nodi), ed essendo questa strettamente positiva[1] un ordinamento ottenuto con la similarity modificata sarà necessariamente uguale ad uno ottenuto con la similarity originale. La formula proposta per il calcolo della similarity è la seguente:

$$\frac{M_{C_1,C_2}}{f} = \left(\sum_{v \in C_1, w \in C_2, v \neq w} A_{vw} \right) - f \left(D_{C_1} D_{C_2} - D^2_{C_1 \cap C_2} \right) \tag{2}$$

Teorema 1. *Le equazioni 1 e 2 sono equivalenti.*

Dimostrazione. Separiamo ed espandiamo le sommatorie:

$$\begin{aligned}
\frac{M_{C_1,C_2}}{f} &= \sum_{v \in C_1, w \in C_2, v \neq w} [A_{vw} - f k_v k_w] = \\
&= \sum_{v \in C_1, w \in C_2, v \neq w} A_{vw} - f \sum_{v \in C_1, w \in C_2, v \neq w} k_v k_w = \\
&= \sum_{v \in C_1, w \in C_2, v \neq w} A_{vw} - f \left(\sum_{i,j} K_{ij} - D^2_{C_1 \cap C_2} \right)
\end{aligned}$$

Dove l'ultima sommatoria è stata trasformata in una sommatoria sugli elementi della matrice K. L'elemento K_{ij}, cioè alla riga i e alla colonna j, è così definito:

$$K_{ij} = k_{v_i} k_{w_j}$$

dove k_{v_i} indica il nodo i-esimo della community C_1, e k_{w_j} il j-esimo nodo della community C_2. Tra le due communities ci possono essere dei nodi a comune[2], e questi saranno gli

[1] In quanto $m > 0$; il caso $m = 0$ non viene analizzato perché giustamente indicherebbe un grafo completamente sconnesso, senza alcun interesse dal punto di vista degli algoritmi di questo tipo.

[2] Addirittura, perché la similarity assuma valore elevato è necessario che ci siano molti nodi a comune!

elementi presenti nella diagonale. La sommatoria originale non considera questi nodi, ed è quindi necessario escluderli sottraendo:

$$D^2_{C_1 \cap C_2} = \sum_{u \in C_1 \cap C_2} k_u$$

al risultato della sommatoria sulla stessa matrice. Il passaggio appena effettuato può sembrare inutile, ma si può fare un notevole raccoglimento nella sommatoria sulla matrice, che ci porta al risultato desiderato:

$$\begin{aligned}
& \textstyle\sum_{i,j} K_{ij} && = \\
= \quad & \textstyle\sum_{i,j} k_i k_j && = \\
= \quad & \textstyle\sum_i \sum_j k_i k_j && = \\
= \quad & \textstyle\sum_i \left(k_i \sum_j k_j \right) && = \\
= \quad & \textstyle\left(\sum_i k_i\right) \left(\sum_j k_j \right) && = \\
= \quad & \textstyle\left(\sum_{v \in C_1} k_v\right) \left(\sum_{w \in C_2} k_w\right) && = \\
= \quad & D_{C_1} D_{C_2}
\end{aligned}$$

□

La formula modificata 2 permette di precalcolare D_{C_1} e D_{C_2}, così da mantenerli almeno per un'iterazione di EAGLE. In realtà questi parametri possono essere mantenuti per tutta la durata di una community, visto che i gradi non cambiano: ogni volta che si crea una community si potrà memorizzare questa quantità, che non cambierà per tutto il tempo di vita di una community[3].

2.3.2 EQ

$$EQ = \frac{1}{2m} \sum_i \sum_{v,w \in C_i} \frac{1}{O_v O_w} \left[A_{vw} - \frac{k_v k_w}{2m} \right]$$

Questo parametro viene valutato di meno rispetto alla similarity: viene infatti calcolato tante volte quante sono le communities iniziali, o più precisamente si calcola una volta per ogni step dell'algoritmo, quindi per ogni set di communities generato.

Il calcolo di questo parametro, che associa ad un set di communities un numero reale, è oneroso. È possibile ottimizzare in svariati modi questa funzione, e ne verranno visti tre in ordine crescente di efficienza, sviluppati tutti sullo stesso ragionamento.

[3] Ancora una volta, l'algoritmo non modifica le connessioni tra i nodi, e quindi i gradi non possono cambiare.

Soft EQ Questa ottimizzazione agisce in due punti:

- rimuove il termine $\frac{1}{2m}$ a moltiplicare, esattamente come fatto per la similarity;
- precalcola per ogni nodo il grado di appartenenza, così da non doverlo calcolare ogni volta che è richiesto; questo da solo è il miglioramento principale applicabile alla formula;

Alla fine la formula ottenuta è la seguente:

$$\frac{EQ}{f} = \sum_i \sum_{v,w \in C_i} \frac{A_{vw} - f k_v k_w}{O_v O_w} \tag{3}$$

L'uguaglianza di questa formula all'originale è dimostrabile in maniera banale, visto che l'unica modifica effettuata alla formula è la rimozione di f. Perché la formula sia valida sempre è però necessario mantenere aggiornati i vari O_v. Questo è un compito che va fatto in due tempi:

- in fase di creazione delle communities: per ogni nodo aggiunto ad una community, il suo grado di appartenenza viene incrementato di uno;
- in fase di merge di due communities.

Quest'ultima fase richiede qualche riga per la spiegazione. Durante la fase di merge due communities C_i e C_j vengono unite per creare la seguente community:

$$C_k = C_i \cap C_j$$

Facciamo un bilancio sul grado di appartenenza dei nodi prima e dopo il merge:

- i nodi che non sono in nessuna community non cambiano grado di appartenenza;
- i nodi che sono solo in una community non cambiano grado di appartenenza, perché escono da una community ma rientrano in un'altra;
- i nodi che sono in entrambe le communities, cioé nella loro intersezione, vedono il loro grado appartenza decrementare di un'unità. Questo perché escono da due community ma rientrano in una soltanto.

Bisogna quindi implementare questa logica nella funzione di merge: il costo computazionale di questo meccanismo è sicuramente trascurabile rispetto ai vantaggi ottenuti utilizzando questo sistema.

Medium EQ Questa ottimizzazione si basa sulla precedente ed è di lieve entità, ed essenzialmente fa una raccolta a fattor comune per $\frac{k_v}{O_v}$. La formula ottenuta è la seguente:

$$\frac{EQ}{f} = \sum_i \sum_{v \in C_i} \frac{k_v}{O_v} \sum_{w \in C_i} \frac{A_{vw} - f k_w}{O_w}$$

In questo caso non è necessario implementare ulteriori meccanismi di consistenza. Si può dimostrare l'equivalenza semplicemente espandendo $\frac{k_v}{O_v}$ all'interno della terza serie.

Hard EQ Questa ottimizzazione si basa ancora sulle precedenti, ma si spinge a livelli estremi; introduce anche un overhead notevole, che riduce di molto il guadagno in termini di prestazioni, ma non lo annulla del tutto. La funzione è la seguente:

$$\frac{EQ}{f} = \sum_i \lambda_i - f\delta_i \tag{4}$$

dove:

$$\lambda_i = \sum_{v,w \in C_i} \frac{A_{vw}}{O_v O_w} \tag{5}$$

è chiamato fattore di connessione interno della community i, mentre:

$$\delta_i = \left(\sum_{v \in C_i} \frac{k_v}{O_v} \right)^2 \tag{6}$$

è chiamato grado pesato della community i.

Teorema 2. *Le funzioni 4 ed 3 sono equivalenti, a meno di una costante moltiplicativa.*

Dimostrazione. Espandiamo la formula 4 con le definizioni 5 e 6 date:

$$\begin{aligned}
& \frac{EQ}{f} & = \\
= \quad & \textstyle\sum_i \sum_{v,w \in C_i} \frac{A_{vw}}{O_v O_w} - f \left(\sum_{v \in C_i} \frac{k_v}{O_v} \right)^2 & = \\
= \quad & \textstyle\sum_i \sum_{v \in C_i} \sum_{w \in C_i} \frac{A_{vw}}{O_v O_w} - f \left(\sum_{v \in C_i} \frac{k_v}{O_v} \right) \left(\sum_{v \in C_i} \frac{k_v}{O_v} \right) & = \\
= \quad & \textstyle\sum_i \sum_{v \in C_i} \sum_{w \in C_i} \frac{A_{vw}}{O_v O_w} - f \sum_{v \in C_i} \left(\frac{k_v}{O_v} \left(\sum_{w \in C_i} \frac{k_w}{O_w} \right) \right) & = \\
= \quad & \textstyle\sum_i \sum_{v \in C_i} \sum_{w \in C_i} \frac{A_{vw}}{O_v O_w} - f \sum_{v \in C_i} \sum_{w \in C_i} \frac{k_v}{O_v} \frac{k_w}{O_w} & = \\
= \quad & \textstyle\sum_i \sum_{v \in C_i} \sum_{w \in C_i} \frac{A_{vw} - k_v k_w}{O_v O_w}
\end{aligned}$$

□

Si vede che l'equazione 4 può essere sfruttata vantaggiosamente: è possibile precalcolare i due termini λ e δ per ogni community. Il problema principale è però il mantenere consistenti questi dati durante le iterazioni, e il costo è elevato. Infatti, oltre alle operazioni indicate per l'aggiornamento del grado di appartenenza, è necessario aggiornare entrambi i termini introdotti per ogni community contenente i nodi il cui grado di appartenenza è cambiato. È evidente che questa non è un'operazione leggera, e il guadagno prestazionale ottenuto con questa soluzione è minimo, seppur presente.

2.3.3 Ordinamento per valori di similarity crescente

Le due modifiche proposte portano a un notevole aumento di velocità di esecuzione dell'algoritmo, in quanto sfruttano l'invarianza di alcuni dati frequentemente usati per tutta l'esecuzione. Viene spontaneo quindi domandarsi se è possibile ottenere un risultato simile anche per la fase di ordinamento, memorizzando in ordine tutti i risultati della similarity, avendo cura di rimuovere i vecchi e di aggiornare i restanti. Esistono diversi metodi per farlo, e in questo progetto sfruttiamo il concetto di lista dei massimi a dimensione limitata, o cache dei massimi.

È infatti impensabile memorizzare tutti i valori di similarity calcolati: supponendo infatti di poter esprimere gli indici delle community su 4B e di utilizzare numeri a virgola mobile a doppia precisione su 8B per esprimere la similarity, si utilizzerebbero 16B solo per la memorizzazione di una tupla $\left(i,\, j,\, M_{C_i,C_j}\right)$. A questi andrebbero poi sommati i byte eventualmente utilizzati dalla struttura dati incaricata di mantenere questi risultati, tenendo presente che avverranno frequenti cancellazioni e inserimenti in qualsiasi posizione:

- per un vettore a dimensione dinamica l'overhead in termini di memoria non dipende dal numero di elementi memorizzati, ma si hanno penalità nel rimuovere gli elementi tanto più pesanti quanto più vicini si è alla testa del vettore[4], e mantenerlo ordinato è ugualmente un'operazione costosa[5]. Questa opzione è sicuramente da scartare, nonostante l'overhead minimo.
- Utilizzando una lista concatenata con un solo puntatore per elemento si ha un overhead di x byte, dove:
$$x = \frac{X}{8}$$
con X il numero di bit utilizzati per l'indirizzamento in memoria. Valori comuni di x sono 4B ed 8B, per sistemi rispettivamente a 32 e 64 bit; consideriamo un sistema con $x = 8B$. L'occupazione totale di memoria per elemento ammonterebbe a 24B, cioè un incremento del 50%. È possibile fare inserimenti e rimozioni con complessità dipendente, sostanzialmente, dalla posizione in cui si vuole inserire o rimuovere un dato; è quindi costoso estrarre o inserire elementi in coda.
- Una lista concatenata con due puntatori per elemento aggiunge un overhead di $2xB$; per un sistema a 64 bit si ha un incremento del 100%, arrivando a 32B. I vantaggi sono comunque notevoli, perché è possibile eseguire operazioni in testa e in coda con complessità costante. Tuttavia, l'inserimento e l'estrazione dei dati in generale hanno complessità lineare, dipendente dalla posizione in cui si vuole inserire o rimuovere un elemento.
- Un insieme ordinato può essere implementato con lo stesso overhead di una lista concatenata, utilizzando una struttura ad albero; verrà però garantito l'inserimento ordinato dei dati con complessità logaritmica (funzione del numero di elementi

[4] Poiché è necessario fare lo shift dei dati per coprire lo spazio lasciato vuoto.

[5] Per lo stesso motivo: un inserimento in testa richiederebbe lo spostamento di tutti gli altri elementi in avanti.

presenti nell'albero), oltre che la rimozione. Nonostante l'overhead in termini di memoria, questa è la struttura dati più veloce che possiamo ottenere.

Utilizzando un insieme per mantenere i valori calcolati abbiamo un costo in memoria di 32B per elemento; gli elementi però sono funzione del numero iniziale di communities, precisamente sono le permutazioni delle communities iniziali:

$$\#elementi = \frac{N(N-1)}{2}$$

Ora, è chiaro che memorizzare tutti i valori è proibitivo:

- grafo IT, con 620 communities iniziali:

$$Mem_{IT} = 191890 * 32B = 6140480B \approx 6MB$$

- grafo Internet, con 36496 communities iniziali:

$$Mem_{Internet} = 665960760 * 32B = 21310744320B \approx 20GB$$

Come già anticipato, la memorizzazione di tutti i dati è impensabile: oltre per l'occupazione in memoria, l'utilizzo di un insieme di dati così grande renderebbe l'algoritmo Memory-Bound.

Si può però utilizzare un approccio diverso, e cioé mantenere una lista limitata dei massimi, chiamata anche cache dei massimi. Nell'implementazione dell'algoritmo abbiamo scelto mantenere al più 3N valori, da ora in poi chiamati massimi. Si memorizza un valore solo se questo ha similarity maggiore del massimo a similarity minima:

$$M_{C_i,C_j} > min\{Cache\}$$

oppure se la cache non ha raggiunto il limite massimo.

Quando si esegue il primo ciclo dell'algoritmo la cache sarà vuota; verrà quindi calcolata la similarity per ogni possibile coppia, e i risultati verranno memorizzati in base al criterio sopra esposto. La cache arriverà ad avere 3N elementi. Si estrae poi l'utilmo elemento dalla cache, cioè quello con valore più alto (arrivando ad avere $3N-1$ elementi), e si fa il merge tra le due community i e j che l'hanno prodotto.

È necessario rimuovere qualsiasi elemento della cache prodotto da una delle due community: non è più presente l'elemento (i, j) in quanto è stato estratto; di conseguenza le due community possono aver generato $N-2$ massimi cadauna. Nel caso massimo, la cache conterrebbe quindi $N+3$ elementi.

Quando si crea una nuova community e la si inserisce nel set è necessario calcolare la similarity tra questa e le altre community. Questo porterà ad avere potenzialmente $N-2$ massimi; se la cache non è vuota questi valori possono essere inseriti direttamente lì dentro, garantendo comunque che la lista mantenga sempre i massimi. Se invece la lista è vuota, è necessario ricalcolare la similarity per tutte le coppie, poiché non abbiamo più informazioni sul vecchio set.

Si può presentare in questo caso una situazione anomala: supponiamo di portarci in una condizione tale da avere un solo elemento (a, b) dopo il merge. Questo elemento da solo è il massimo tra tutti gli altri rimasti nel set. Quando inseriamo la nuova community può succedere che nessun elemento sia maggiore dell'elemento in questione; tuttavia per il criterio della cache questi elementi entrano *tutti* nella lista. In seguito è ovvio che verrà estratto l'elemento di massimo valore, cioè (a, b). Verranno inoltre invalidati sempre e solo due elementi dalla cache. Ora la lista dei massimi si è ristretta ad un sottoinsieme dello spazio delle soluzioni: quando verranno generati i massimi della nuova community noi non possiamo sapere se nel vecchio set c'era qualche valore minore di (a, b) ma maggiore di qualsiasi altro elemento aggiunto in seguito nella cache. Non sono state trovate (ma non se ne esclude l'esistenza) relazioni sulla similarity che garantiscono la presenza di questo massimo nei nuovi valori generati, oppure che una situazione simile non si può presentare. Di conseguenza, si aggiunge un'ulteriore regola: se il valore massimo presente in cache prima e dopo l'aggiornamento della cache dopo un merge è lo stesso, la cache è da invalidare completamente. Va quindi svuotata, e ripopolata all'inizio dell'iterazione successiva.

Nonostante tutte queste complicazioni, il guadagno è notevole. Chiaramente, incrementando la dimensione massima della cache la probabilità di invalidare tutta la cache è diventa bassa, ma si rischia di utilizzare troppa memoria.

2.4 Implementazione (federico)

L'implementazione dell'algoritmo EAGLE è stata realizzata in C++, con l'utilizzo delle librerie *igraph* e *boost*. Tali librerie sono opensource.