DIPARTIMENTO DI
INGEGNERIA INFORMATICA,
MODELLISTICA, ELETTRONICA
E SISTEMISTICA
DIMES

Corso di Basi di Dati Evolute e Data Mining

Progetto "Influence Maximization with Diversity Measures"

Studenti:

Domenico Costantino matr. 189168

Giovanni Bruno matr. 189145

Professore:

Andrea Tagarelli

Anno Accademico 2017/2018

Sommario

Introduzione

Viviamo in comunità e interagiamo con i nostri amici, familiari e persino con gli estranei. Condividiamo idee e conoscenze con loro. Nel processo, ci influenziamo a vicenda. Questa influenza sociale, se modellata correttamente, può essere sfruttata in varie applicazioni come Viral Marketing, Recommender Systems, Feed Ranking ecc.

Uno dei problemi fondamentali è l'identificazione di clienti influenti nelle reti sociali. Questi se convinti ad adottare un prodotto, lo sponsorizzeranno tra i loro amici e porteranno a un gran numero di adozioni, spinti dall'effetto passaparola (marketing virale).

Il problema della massimizzazione dell'influenza è definito come segue. Dato un grafo sociale con probabilità di influenza sugli archi e un numero *k*, l'obiettivo è selezionare *k* nodi seed in modo tale che attivandoli, la diffusione di influenza prevista sia massimizzata.

In particolare, il lavoro seminale di Kempe si concentra su due modelli di propagazione di base. [1]

- Linear Threshold Model
- Independent Cascade Model

Modello Linear Threshold e funzioni submodulari

Considerando una rete sociale rappresentata da un grafo diretto *G*, i cui nodi possono essere *attivi*, cioè promotori della diffusione, oppure *inattivi*, ci si basa sull'assunzione che la tendenza di un nodo ad attivarsi cresca monotonamente col numero di vicini attivi. Il modello Linear Threshold è progressivo: prende in considerazione solo il fenomeno dell'attivazione. Non è quindi possibile che un nodo attivo si disattivi. È importante considerare che un nodo v è influenzato da ogni vicino w con un peso $b_{v,w}$ con la regola che:

$$\sum_{w\ neighbor\ of\ v} b_{v,w} \leq 1$$

Ogni nodo ha una propria *soglia di attivazione* θ compresa nell'intervallo [0,1], che rappresenta la tendenza del nodo a farsi influenzare. Se la somma delle influenze dei vicini supera la soglia il nodo si considera attivato.

Dato un insieme di nodi attivi iniziali il processo di diffusione si sviluppa attraverso una sequenza di passi discreti e deterministici. C'è inoltre da considerare che un nodo che è stato attivato al passo $t-1$ resta attivo al passo t.

Nel loro importante lavoro del 2003 Kempe, Kleinberg e Tardos, oltre a definire i modelli, hanno fornito delle approssimazioni possibili, con dei bound relativi agli errori possibili.

Partiamo con la definizione di funzione *submodulare*. Considerando una funzione f che associa un numero reale non negativo ad un insieme S, intuitivamente possiamo dire che essa è submodulare se il guadagno marginale relativo all'aggiunta di un elemento ad S è al più grande quanto il guadagno marginale ricavato nell'aggiungere lo stesso elemento a un superset di S. Formalmente, una funzione submodulare soddisfa la relazione:

$$f(S \cup \{v\}) - f(S) \geq f(T \cup \{v\}) - f(T)$$

per tutti gli elementi v a le coppie di insiemi $S \subseteq T$.

Le funzioni submodulari godono di varie proprietà apprezzabili ma quella rilevante nel nostro caso è la seguente. Supponendo di avere una funzione f che restituisca solo valori non negativi e che sia monotona nel senso che aggiungendo elementi all'insieme essa non diminuisca, si vuole individuare un insieme di k elementi per Il quale $f(S)$ sia massima.
Questo è un problema di ottimizzazione NP-hard, ma si è riusciti a trovare un algoritmo greedy hill-climbing che approssima l'ottimo con un fattore $(1 - 1/e)$.

Dato che nel caso della diffusione in esame è molto oneroso (#P-hard), e quindi nella maggior parte dei casi impossibile riuscire a calcolare il marginal gain in maniera esatta, l'ottimizzazione compie un errore dell'ordine di $(1 - 1/e - \varepsilon)$, con $\varepsilon > 0$ e dipendente da quanto si vuole approssimare nel calcolo dei guadagni. [2]

L'algoritmo Simpath

Ciò che si vuole ricavare è un insieme di nodi (*seed set*), che massimizzi lo *spread* totale della rete, definito come la somma dell'influenza di nodi attivi verso quelli inattivi. Dato che l'obiettivo non è massimizzare il numero di nodi attivi si possono mettere da parte le soglie dei nodi del modello LT, considerando però il limite di 1 alla sommatoria dei pesi relative alle influenze dei vicini per ogni nodo. Negli anni si sono sviluppati vari approcci.

Simple Greedy algorithm: In ogni iterazione, si aggiunge al seed set, il nodo che fornisce il guadagno marginale massimo in termini di spread. È un algoritmo semplice e con una approssimazione dimostrata ma è anche oneroso dato che calcolare il guadagno marginale ha costo #P e vengono effettuate O (n * k) chiamate dell'algoritmo che lo stima.

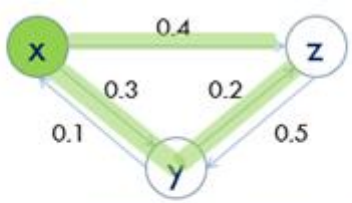
LDAG: Calcolare lo spread in grafi generali, come visto, è #P. Ma si può calcolare in tempo lineare su DAGs (Directed Acyclic Graphs). La maggior parte dell'influenza di un nodo è diffusa in un piccolo vicinato, quindi per ogni nodo si costruisce un DAG locale e si considera solo l'influenza diffusa in quel LDAG. L'algoritmo si basa molto sulla ricerca di buoni LDAG. Trovare un LDAG ottimale è NP-hard. Si usa un'euristica greedy ma non è fornita nessuna garanzia di approssimazione. Inoltre, l'algoritmo considera lo spread di influenza da un solo DAG locale e ignora altri DAG. Se lo

spread da altri DAG locali è significativo, le prestazioni potrebbero essere scadenti. Dato che si mantiene un DAG per nodo, il consumo di memoria è elevato.

Si può considerare **Simpath** come un algoritmo greedy con una logica "lazy forward", cioè che, ad ogni iterazione, aggiunge al seed set il nodo che fornisce il guadagno marginale massimo in spread. Vediamo come SIMPATH-SPREAD stima lo spread di un nodo.

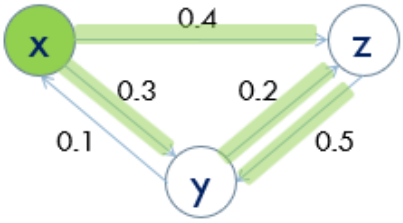
Si osservi che l'influenza di un nodo x sul nodo z può essere calcolata enumerando tutti i percorsi semplici che iniziano da x e finiscono in z.

Un percorso semplice è un percorso che non contiene cicli ed il cui valore è calcolato moltiplicando i pesi degli archi attraversati.



$$\Upsilon_{x,z} = 0.3 \cdot 0.2 + 0.4 = 0.46$$

Lo spread implicito di un nodo (verso sé stesso) è sempre pari a 1.



Influence of x on z
Influence of x on x itself
Influence of x on y
Total influence of node x is 1.96

Influence Spread of node x is $\Upsilon_{x,x} + \Upsilon_{x,y} + \Upsilon_{x,z}$

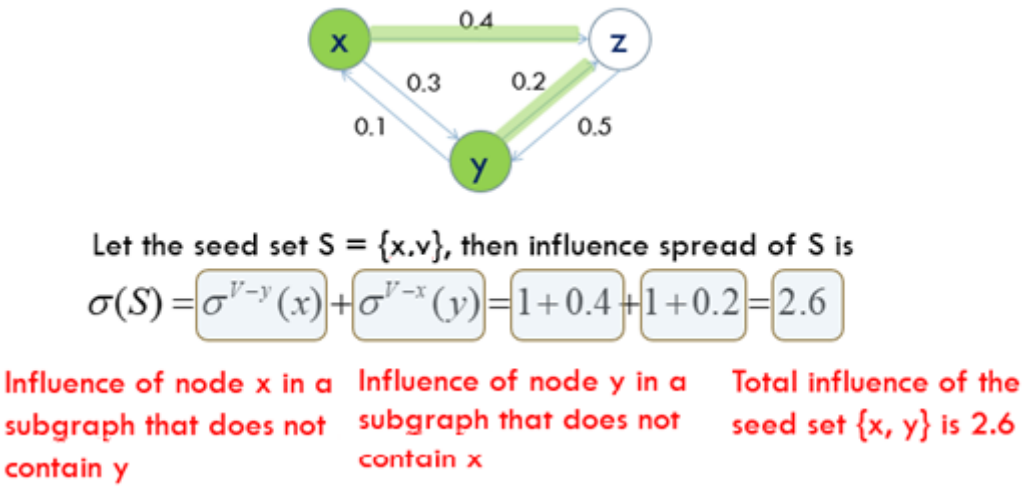
Fondamentale è il risultato del seguente teorema.

Teorema 1. Nel modello LT, lo spread di un set S è la somma dello spread di ogni nodo $u \in S$ sui sottografi indotti da $V - S + u$, ovvero

$$\sigma(S) = \sum_{u \in S} \sigma^{V-S+u}(u)$$

Quindi, l'influenza può essere stimata enumerando tutti i percorsi semplici a partire dal seed set in un piccolo vicinato. Attraverso un parametro η possiamo controllare la grandezza del vicinato (smettendo di enumerare i percorsi quando il peso dell'influenza scende sotto η), ciò permette di avere un trade-off tra tempo di esecuzione e prestazioni. Per enumerare i singoli path si usa un algoritmo

classico di backtracking. La seguente immagine è un esempio del calcolo dello spread, applicando il Teorema 1.



Simpath richiede quindi una chiamata di SIMPATH-SPREAD per calcolare il guadagno marginale di ogni nodo. Questo approccio non è per niente scalabile quindi si è pensato a due ottimizzazioni, che verranno esposte nel seguito, utili a ridurre il tempo richiesto.

SIMPATH-SPREAD è l'algoritmo che permette di stimare lo spread di insieme *S* relativamente all'insieme dei nodi *V* del grafo in input (vedi Algorithm 1). Per enumerare tutti i possibili path, per ogni nodo di *S* si effettua una chiamata dell'algoritmo *Backtrack* che calcola ricorsivamente tutti i cammini semplici che si generano a partire dal nodo in esame (vedi Algorithm 2).

Algorithm 1 SIMPATH-SPREAD

Input: S, η, U
1: $\sigma(S) = 0$.
2: **for** each $u \in S$ **do**
3: $\quad \sigma(S) \leftarrow \sigma(S) + \text{BACKTRACK}(u, \eta, V - S + u, U)$.
4: return $\sigma(S)$.

Algorithm 2 BACKTRACK

Input: u, η, W, U
1: $Q \leftarrow \{u\}$; $spd \leftarrow 1$; $pp \leftarrow 1$; $D \leftarrow null$.
2: **while** $Q \neq \emptyset$ **do**
3: $\quad [Q, D, spd, pp] \leftarrow \text{FORWARD}(Q, D, spd, pp, \eta, W, U)$.
4: $\quad u \leftarrow Q.last()$; $Q \leftarrow Q - u$; $delete\ D[u]$; $v \leftarrow Q.last()$.
5: $\quad pp \leftarrow pp/b_{v,u}$.
6: return spd.

A sua volta il backtracking richiama il metodo *Forward* che calcola lo spread generato dal cammino semplice. Le righe numero 9 e 10 dell'Algorithm 3 concretizzano l'ottimizzazione "Look Ahead" e "vertex cover" che verranno esposte nel seguito.

Algorithm 3 FORWARD

Input: $Q, D, spd, pp, \eta, W, U$

1: $x = Q.last()$.
2: **while** $\exists y \in N^{out}(x)$: $y \notin Q, y \notin D[x], y \in W$ **do**
3: **if** $pp \cdot b_{x,y} < \eta$ **then**
4: $D[x].insert(y)$.
5: **else**
6: $Q.add(y)$.
7: $pp \leftarrow pp \cdot b_{x,y}$; $spd \leftarrow spd + pp$.
8: $D[x].insert(y)$; $x \leftarrow Q.last()$.
9: **for** each $v \in U$ such that $v \notin Q$ **do**
10: $spd^{W-v} \leftarrow spd^{W-v} + pp$.
11: return $[Q, D, spd, pp]$.

Per quanto riguarda SIMPATH, esso è composta da due fasi:

- La prima in cui viene richiamato per ogni nodo del vertex cover del grafo l'algoritmo SIMPATH-SPREAD. Questo permette di popolare la cosiddetta "CELF queue", una coda prioritaria contente tutti i nodi del grafo e pesata rispetto ad uno score, che nel caso di Simpath è lo spread generato dal nodo. Grazie all'ottimizzazione Look Ahead si riesce a ridurre di molto le chiamate, in quanto per i nodi non presenti nel vertex cover lo spread si calcola in tempo costante senza chiamate ricorsive (righe 1-8 di Algorithm 4).
- Nella seconda, partendo dalla stima dello spread dei nodi nella CELF queue, ed utilizzando ancora una volta l'ottimizzazione Look Ahead, si calcola il guadagno marginale di ogni nodo, in termini di spread, rispetto a quelli già presenti nel seed set (inizialmente vuoto). Ad ogni iterazione si seleziona in maniera greedy il nodo con guadagno marginale più alto (righe 9-20 di Algorithm 4). [2]

Algorithm 4 SIMPATH

Input: $G = (V, E, b), \eta, \ell$

1: Find the vertex cover of input graph G. Call it C.
2: **for** each $u \in C$ **do**
3: $U \leftarrow (V - C) \cap N^{in}(u)$.
4: Compute $\sigma(u)$ and $\sigma^{V-v}(u)$, $\forall v \in U$ in a single call to Algorithm 1: SIMPATH-SPREAD(u, η, U).
5: Add u to CELF queue.
6: **for** each $v \in V - C$ **do**
7: Compute $\sigma(v)$ using Theorem 2.
8: Add v to CELF queue.
9: $S \leftarrow \emptyset$. $spd \leftarrow 0$.
10: **while** $|S| < k$ **do**
11: $U \leftarrow$ top-ℓ nodes in CELF queue.
12: Compute $\sigma^{V-x}(S)$, $\forall x \in U$, in a single call to Algorithm 1: SIMPATH-SPREAD(S, η, U).
13: **for** each $x \in U$ **do**
14: **if** x is previously examined in the current iteration **then**
15: $S \leftarrow S + x$; Update spd.
16: Remove x from CELF queue. Break out of the loop.
17: Call BACKTRACK$(x, \eta, V - S, \emptyset)$ to compute $\sigma^{V-S}(x)$.
18: Compute $\sigma(S + x)$ using Eq. 6.
19: Compute marginal gain of u as $\sigma(S + x) - spd$.
20: Re-insert u in CELF queue such that its order is maintained.
21: return S.

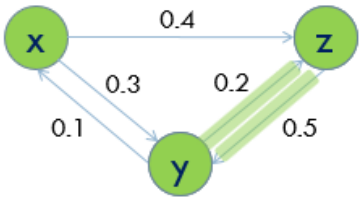
Vertex Cover Optimization

L'ottimizzazione Vertex Cover migliora il tempo di esecuzione della prima iterazione, risolvendo in tal modo la debolezza principale dell'ottimizzazione CELF, su cui Simpath si basa. Goyal, nel documento, si avvale del seguente,

Teorema 2. Nel modello LT, lo spread di un nodo dipende linearmente da quello dei suoi out-neighbors, nel seguente modo,

$$\sigma(v) = 1 + \sum_{u \in N^{out}(v)} b_{v,u} \cdot \sigma^{V-v}(u)$$

Ciò permette di calcolare "direttamente" lo spread di un nodo usando lo spread dei suoi vicini esterni. Quindi, nella prima iterazione, si costruisce il vertex cover *C* e si calcola lo spread solo per i nodi in *C*. Lo spread degli altri nodi può essere calcolata "direttamente", in tempo costante. Di seguito un esempio. [2]



$$\sigma(x) = 1 + b_{x,y} * \sigma^{V-x}(y) + b_{x,z} * \sigma^{V-x}(z)$$
$$= 1 + 0.3 * (1 + 0.2) + 0.4 * (1 + 0.5) = 1.96$$

Look Ahead Optimization

All'aumentare del seed set, aumenta il tempo speso per stimare lo spread (più percorsi da enumerare). Tuttavia, molti percorsi vengono ripetuti. L'ottimizzazione evita questa ripetizione in modo intelligente, mediante un parametro di look ahead *l*.

In una iterazione, l'ottimizzazione utilizza i top-*l* elementi dalla coda CELF e calcola lo spread del seed set *S* sui grafi indotti $V - x$ per tutti i nodi (top-*l*). Quindi calcola lo spread di *x* sul grafo $V - S_i$. Manipolando il Teorema 1 si ottiene lo spread di $S_i + x$, nel seguente modo.

$$\sigma(S_i + x) = \sum_{u \in S_i + x} \sigma^{V - S_i - x + u}(u)$$

$$= \sigma^{V - S_i}(x) + \sum_{u \in S_i} \sigma^{V - S_i - x + u}(u)$$

$$= \sigma^{V - S_i}(x) + \sigma^{V - x}(S_i)$$

Questo permette di stimare il marginal gain di x rispetto ad S_i, e di aggiornare lo score di x nella CELF queue con questo valore. Se uno dei seed riappare, in una iterazione successiva, tra i top-*l* ci fermiamo e lo inseriamo nel seed set, altrimenti vengono presi i top-*l* elementi successivi. *l* = 1 non implica ottimizzazione. [2]

Definizione di Diversità

Nell'ambito dei big data assume particolare rilevanza il concetto di diversità, sia per motivi etici, cioè limitare il rischio di discriminazioni, che pratici, per rendere i metodi che analizzano i comportamenti umani più accurati.

Con diversità si intende una misura atta a descrivere il grado di eterogeneità di un oggetto composto, il quale è solitamente il risultato di un processo algoritmico.

La diversità può essere intesa in modi differenti in base al contesto applicativo. Si può definire il problema della diversificazione come un problema di selezione. Data una misura di diversità *div* e un set *I* di *n* elementi, e un intero *k*, *k*<=*n*, si vuole ricavare un insieme *S* di *k* elementi di *I*, tale che

$$S = \underset{S' \subseteq I,\ |S'| = k}{argmax} div(S')$$

Bisogna però definire adeguatamente cosa si intente per *div*. Nel nostro caso, il set da considerare è formato da nodi caratterizzati da attributi categorici che si cerca di diversificare. La misura di diversità di un determinato insieme è definita come la combinazione lineare delle diversità relative a tutti gli attributi da considerare.

$$div(S) = \alpha \cdot div_{A1}(S) + \beta \cdot div_{A2}(S) + \ldots + \gamma \cdot div_{An}(S)$$

con $\alpha + \beta + \ldots + \gamma = 1$ e $A1, A2, \ldots, An$ attributi dei nodi di *S*. Ciascuno degli Ai $(i = 1..n)$ ha un diverso dominio di valori.

Proprietà rilevanti che dovrebbe avere una funzione di diversità sono la linearità e la submodularità così definita:

$$f(S + \{w\}) - f(S) \geq f(T + \{w\}) - f(T) \quad con\ S \subseteq T\ e\ w \notin T$$

A questo punto si deve definire cosa si intende per $div_X(S)$, per ogni attributo X. Si sono fatte varie considerazioni su cosa significasse il grado di diversità di un insieme di elementi rispetto ad un attributo comune ma che può assumere valori diversi, e si è arrivati alla conclusione che un modo calzante per definirlo è il seguente:

$$div_X(S) = \frac{\#valoriDistinti(X,S)}{|S|}$$

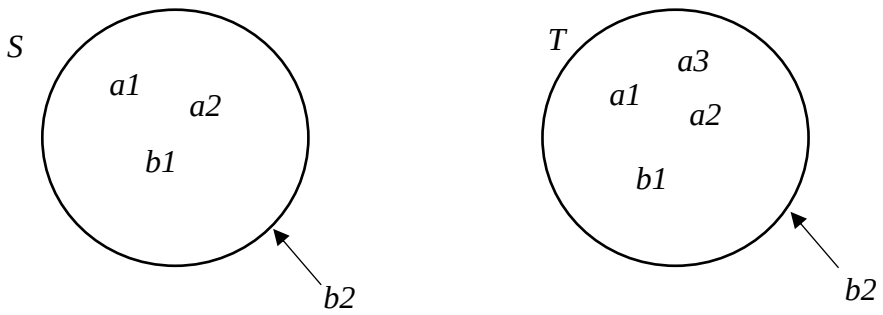
"il numero di valori diversi che assume l'attributo *X* all'interno dell'insieme *S*, normalizzato rispetto alla cardinalità dell'insieme stesso".

Il principale problema è che questa misura, anche essendo rappresentativa della realtà, non gode della proprietà di monotonicità (dato che aggiungendo nodi con lo stesso valore dell'attributo *X* si avrà un decremento e aggiungendo nodi con valori diversi un incremento), e tantomeno di quella, desiderata, di submodularità.

Si è quindi considerata una misura basata sulla distanza tra nodi, come spiegato nel documento. In particolare, in questa concezione, la diversità di un insieme è la media delle distanze tra coppie di nodi dell'insieme.

$$div_X(S) = \frac{1}{|S|}\sum_{i,j\in S, i\neq j} d_X(i,j) \qquad con \quad \begin{cases} d_X(i,j) = 1 & se\ val_X(i) \neq val_X(j) \\ d_X(i,j) = 0 & se\ val_X(i) = val_X(j) \end{cases}$$

Anche in questo caso non è verificata la submodularità perché considerando il seguente esempio (un solo attributo, X, con valori *a* e *b*, e numerazione relativa ai nodi):



Si noti che $S \subseteq T$ e a_i indica un nodo che nell'attributo *X* assume valore *a*. Le possibili coppie, per cui vale $d_X(i,j) = 1$ sono {*(a1,b1),(a2,b1)*} per *S* e {{*(a1,b1),(a2,b1),(a3,b1)*} per *T*. Andando a verificare la submodularità si avrà:

$$div_x(S) = \frac{1+1}{3} = \frac{2}{3} \qquad div_x(S + \{b2\}) = \frac{1+1+1+1}{4} = 1$$

$$div_x(T) = \frac{1+1+1}{4} = \frac{3}{4} \qquad div_x(T + \{b2\}) = \frac{1+1+1+1+1+1}{5} = \frac{6}{5}$$

$$div_x(S + \{b2\}) - div_x(S) \geq div_x(T + \{b2\}) - div_x(T)$$

$$1 - \frac{2}{3} \geq \frac{6}{5} - \frac{3}{4} \qquad \rightarrow \qquad \frac{1}{3} \geq \frac{9}{20}$$

Si è quindi appurato che questa misura di diversità non è submodulare. Inoltre, è particolarmente dispendiosa da calcolare all'aumentare della taglia dell'insieme (cresce in maniera quadratica).

Ci si è quindi concentrati sul cercare di perseguire la submodularità, modificando la concezione di diversità. Un primo approccio (naïve) è stato quello di considerare solo i valori dell'attributo già presenti nell'insieme e aumentare il valore di diversità di 1 quando si va ad inserire un nodo con valore di attributo diverso (V_x sta per dominio dei valori di *X*):

$$div_x(S) = \sum_{a \in V_x} contains(a,S) \ \ con \begin{cases} contains(a,S) = 1 & se \ \exists nodo \ i \in S \ che \ ha \ valore \ di \ X \ pari \ ad \ a \\ contains(a,S) = 0 & altrimenti \end{cases}$$

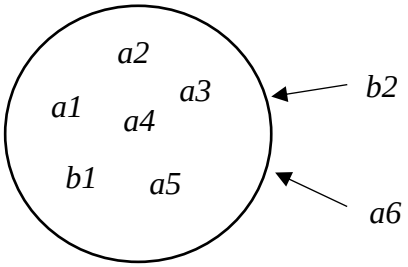
Questa misura è monotona dato che può solo crescere al crescere della taglia dell'insieme.

È anche submodulare. Dato che $S \subseteq T$ se il nuovo nodo che vogliamo inserire ha un valore di attributo già contenuto in *S* sarà anche già presente in *T* quindi la relazione della submodularità si riduce a $0 \geq 0$. Se il nodo da inserire ha un valore di attributo non presente in *S* si avrà la situazione

$$1 \geq v \qquad con \begin{cases} v = 1 & se \ il \ valore \ del \ nodo \ non \ è \ presente \ in \ T \\ v = 0 & se \ il \ valore \ del \ nodo \ è \ presente \ in \ T \end{cases}$$

In entrambi i casi la relazione è verificata.

Questa misura non permette però di tenere in considerazione le ripetizioni di valori di attributo, ossia situazioni del tipo



In cui il contributo alla diversità dell'insieme (marginal gain) che potrebbero apportare *a6* e *b2* viene considerato uguale (0 per entrambi). Questa situazione però non è molto in accordo con il concetto che si vuole rappresentare. Si dovrebbe fare in modo che il contributo di *b2* sia maggiore di quello di *a6*, si è quindi arrivati ad una nuova formulazione.

La funzione di valutazione $div_x(S)$, nel nostro caso, è definita come: chiamiamo V_x l'insieme di valori che assume l'attributo *x* all'interno del set *S*, e *num(v)* il numero di occorrenze di nodi che hanno il valore $v \in V_x$, si avrà

$$div_x(S) = \sum_{v \in V_x} \sum_{i=1}^{num(v)} \frac{1}{i}$$

Si può notare che l'argomento della prima sommatoria è una serie armonica relativa alle occorrenze di *v*. In particolare, il valore marginale ottenuto dall'inserimento di un nuovo nodo all'interno del set sarà pari al contributo *num(v)*+1-esimo della relativa serie.

Si può dimostrare che questa funzione gode sia della proprietà di monotonicità che di quella di submodularità. La dimostrazione della prima è banale, per la seconda si può considerare il caso generale della definizione e osservare che:

$$f(S + \{w\}) - f(S) \geq f(T + \{w\}) - f(T) \quad con\ S \subseteq T\ e\ w \notin T$$

Nel nostro caso si avrà $f(S + \{w\}) - f(S) = \frac{1}{\alpha}$, ovvero α sarà la cardinalità del sottoinsieme S' di S tale che $\forall nodo\ n \in S',\ val_x(n) = val_x(w)$. Come detto prima $\frac{1}{\alpha}$ è proprio il marginal gain ottenuto dall'inserimento di *w* in *S*.

Allo stesso modo $f(T + \{w\}) - f(T) = \frac{1}{\beta}$ con β la cardinalità del sottoinsieme T' di T tale che $\forall nodo\ n \in T',\ val_x(n) = val_x(w)$.

Dato che vale $S \subseteq T$, $\beta \geq \alpha$, quindi $\frac{1}{\alpha} \geq \frac{1}{\beta}$ e la relazione sarà verificata.

Il valore per quest'ultima formulazione può essere calcolato in maniera abbastanza veloce e, cosa importante, si può calcolare il guadagno marginale di un nodo senza andare a ricalcolare il valore di diversità per tutto l'insieme, rendendo il costo della computazione pressoché costante.

Una possibile estensione potrebbe essere quella di usare una serie geometrica con argomento $\frac{1}{e}$ (oppure $\frac{1}{a}$, con *a* costante) mitigando ancora il contributo in caso di valori già presenti nell'insieme. In questo caso si può definire:

$$div_x(S) = \sum_{v \in V_x} \sum_{i=1}^{num(v)} \frac{1}{e^i}$$

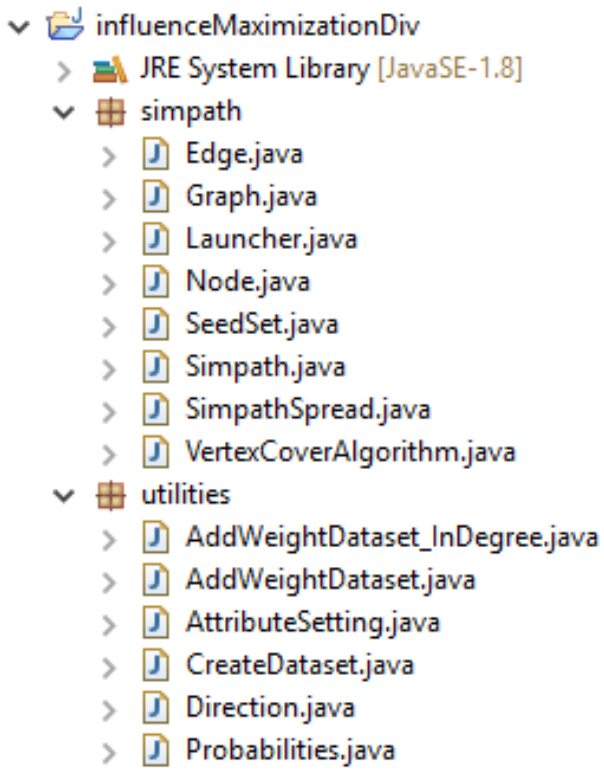
La misura di diversità complessiva ottenuta può essere aggregata a sua volta con la misura di marginal gain dello spread, ottenendo così uno score utilizzato nella priority queue dell'algoritmo simpath.

$$score(S) = \alpha \cdot spread(S) + (1 - \alpha) \cdot div(S)$$

con $\alpha \in [0,1]$.

Struttura del codice e cenni implementativi

La nostra implementazione è composta da due packages. Nel primo, "simpath" è contenuta la concretizzazione dell'algoritmo di influence maximization che tiene conto, però, anche della misura di diversità. Il package "utilities" è invece utile a configurare opportunamente il grafo in input, in modo che l'algoritmo possa operarvi correttamente. La struttura delle classi che appaiono nel progetto è la seguente.



È importante sottolineare che per la corretta esecuzione è necessario che sia installata sulla macchina la versione 8 di Java, dato che nel progetto, in più punti, si fa uso di classi e costrutti (la classe "Recursive Task" per la parallelizzazione di SIMPATH-SPRED, oppure il casting implicito di strutture che utilizzano i generics) introdotti in questa versione.

Il package “utilities”

Le classi di questo package nascono dal bisogno di generare un input utile a testare la correttezza dell’algoritmo. La maggior parte dei datasets presenti in rete rappresentano delle reti sociali definite su grafi orientati, ma non pesati. Questa caratteristica è fondamentale nel nostro caso, dato che il peso associato ad ogni arco (*i*,*j*) misura l’influenza che il nodo *i* ha sul nodo *j*.

In questo senso state predisposte due classi.

- *AddWeightDataset* che permette, tramite un oggetto di classe *Random* di Java di attribuire un peso compreso tra 0 e 0.5 ad ogni arco del dataset, facendo un modo che la relazione per la quale la somma dei pesi degli archi entranti in ogni nodo sia minore o uguale a 1, sia soddisfatta.
- *AddWeightDataset_InDegree* è molto simile alla classe precedente ma tiene conto, durante l’attribuzione del peso, dell’in-degree del nodo destinazione, attribuendo un valore inversamente proporzionale al grado. Questo permette di concretizzare l’assunzione che, data una entità che è in contatto con molte altre, è più difficile che una sola di queste riesca a dare un contributo determinante nel convincere l’entità considerata.

Entrambe le classi producono un nuovo file di testo del tipo <nomefile>+”Weighted.txt”, con <nomefile> il nome del file di testo in input da cui leggere l’insieme di archi del grafo.

La classe *CreateDataset* permette di creare dei grafi orientati, pesati e con degli attributi sui nodi, di piccole dimensioni, ed è stata utile nella fase di testing.

Un altro problema con il quale ci siamo scontrati è stato quello di assegnare degli attributi categorici ad ogni nodo del grafo, sui quali poi basare il calcolo della diversità. La classe *AttributeSetting* ha lo scopo di generare un nuovo file in output in cui, dato un insieme di nodi di ingresso si associa ad ognuno un insieme di valori di attributo, in accordo a varie distribuzioni di probabilità sui domini di valori di ogni attributo. I possibili valori di ogni attributo, il numero di attributi e la scelta su quale distribuzione utilizzare vengono forniti in input, da console, a runtime dall’utente.

La classe *Probabilities* permette di associare, per ogni dominio di ogni attributo un valore ad ogni nodo, in accordo alla distribuzione di probabilità scelta dall’utente in input. Le scelte possono essere 3,

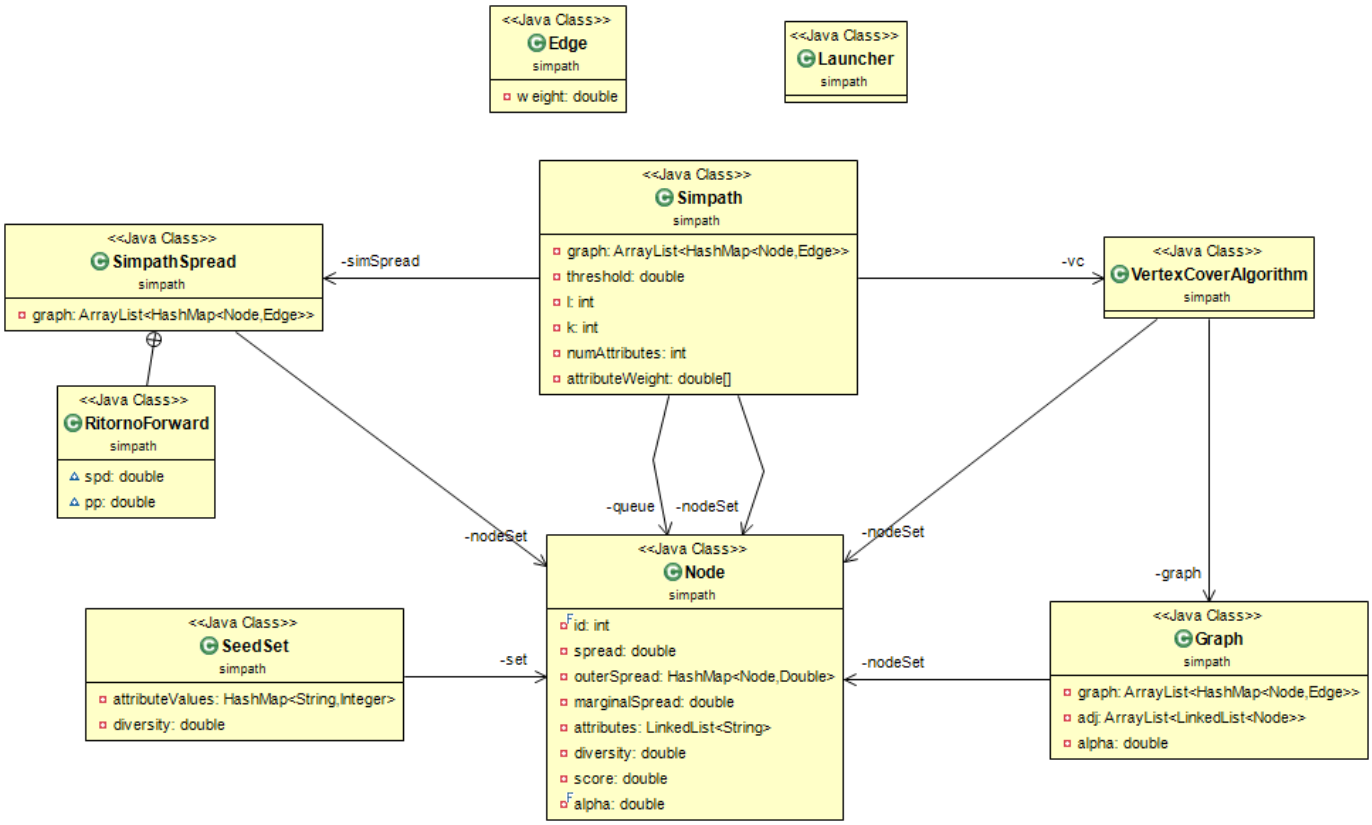
- A priori: ovvero le probabilità con la quale può comparire ogni valore viene fornita in input dall’utente quando si specificano i valori possibili. Ovviamente la somma delle probabilità per ogni attributo deve essere 1.
- Uniforme: il valore dell’attributo viene scelto in maniera uniforme tra quelli possibili.
- Esponenziale di parametro λ: si attribuisce una probabilità più alta ai valori specificati per primi, in base al valore, specificato anch’esso in input, $\lambda \in [0,1]$.

Ogni metodo di Probabilities, che implementa un diverso concetto di distribuzione, per simulare l'esito dell'esperimento aleatorio della scelta del valore, fa uso del metodo *nextDouble* della classe Random fornita da Java.

Avvalendosi di dei metodi di utilità di *Probabilities* la classe *AttributeSetting* genera un file nella forma <nomefile>+"Attr.txt", con <nomefile> il nome del file di testo in input da cui leggere l'insieme di nodi del grafo, in cui sono presenti i valori di ogni attributo associati ad ogni nodo della rete in input, permettendo così di effettuare più test sulla stessa configurazione (nodi, collegamenti, influenza e attributi categorici) variando i pesi associati alla diversità e allo spread (oppure i pesi relativi ai vari attributi) e di poter confrontare i risultati.

Il package "simpath"

È organizzato nel seguente modo.



Le classi contenute nel package hanno lo scopo di concretizzare l'algoritmo di influence maximization che tiene conto della diversità, utilizzando le ottimizzazioni presenti nel documento di Goyal. Nel dettaglio:

La classe *Node* rappresenta il concetto di nodo e contiene tutte le strutture fondamentali per calcolare lo score su cui si basa la priority queue di SIMPATH, le variabili *outerSpread* e *marginalSpread* servono per implementare l'ottimizzazione Look Ahead. *Graph* invece rappresenta il grafo in input e contiene, con opportune strutture, l'informazione riguardante i nodi presenti e gli archi che li connettono, e consente di leggere il grafo dal file in input e di stamparlo.

Il cuore del package sono le classi *Simpath, SimpathSpread* e *VertexCover* con concretizzano l'algoritmo per come è presente nel documento del 2011.

In particolare, il vertex cover si basa sull'euristica maximum degree, dato che calcolare il vertex cover esatto è notoriamente difficile e oneroso. La nostra implementazione crea una copertura scegliendo iterativamente i nodi di grado massimo ed eliminando dal grafo i nodi (e quindi gli archi che li connettono) già coperti.

La classe *Simpath* si avvale di un oggetto *SeedSet* che rappresenta il risultato della computazione e che viene costruito dall'algoritmo. Esso permette in tempo costante, data la nostra definizione, di stimare il guadagno marginale relativo alla diversità di un nodo rispetto a questo insieme.

La classe *SimpathSpread* concretizza la valutazione dello spread di un insieme come la somma dello spread dei suoi nodi, valutando ricorsivamente i cammini semplici che si originano da ogni nodo. Per rendere più veloce la valutazione si è pensato di parallelizzarla utilizzando la classe RecursiveTask di Java 8. Ciò ha reso l'algoritmo più veloce e scalabile nel caso di datasets corposi. La versione multithreaded dell'algoritmo si è dimostrata all'incirca veloce il doppio rispetto a una single thread.

La classe *Launcher* ha lo scopo di interfacciarsi con l'utente e di prendere in output il file relativo al grafo pesato, di ricevere dall'utente tutti i parametri necessari, come α (peso dello spread rispetto alla diversità), i pesi associati ai vari attributi, e la scelta tra la versione single o multithreaded. All'occorrenza può richiamare la classe *AttributeSetting* per cambiare gli attributi e i valori assegnati ad ogni nodo del grafo. La classe avvia l'algoritmo e stampa a schermo il risultato.

Per tutti i dettagli implementativi si rimanda al codice allegato.

Risultati ottenuti

Durante lo sviluppo sono stati fatti vari test su input di diverse dimensioni. Per un testing più realistico si è scelto di generare dei grafi orientati che rappresentassero più fedelmente delle reti sociali. A questo proposito, utilizzando RStudio, si è pensato al modello Barabasi-Albert per la generazione della rete.

EVIDENZIARE CARATTERISTICHE MODELLO

Il principale problema riscontrato è che, per come è definito il modello, il grado di uscita dei nodi è costante; quello che varia è il grado di ingresso. Dato che per i nostri scopi è rilevante avere delle reti in cui sono presenti degli influencers (con grado di uscita elevato) si è pensato di invertire la direzione degli archi generati dal Barabasi-Albert ottenendo un dataset adeguato. Questa operazione viene

svolta nella classe *AddWeightDataset_InDegree* del package "utilities" e genera un nuovo file di testo nella forma <nomefile>+"Weighted.txt", con <nomefile> il nome del file di testo in input da cui leggere l'insieme di archi del grafo generato con RStudio.

I vari test effettuati considerano grafi di dimensione sempre maggiore.

TEST 1. Il grafo in questione chiamato "littlegraph", ricavato con il metodo di Barabasi-Albert, è costituito da 200 nodi e [scrivere numero archi] . Si sono considerati un insieme di attributi così composto:

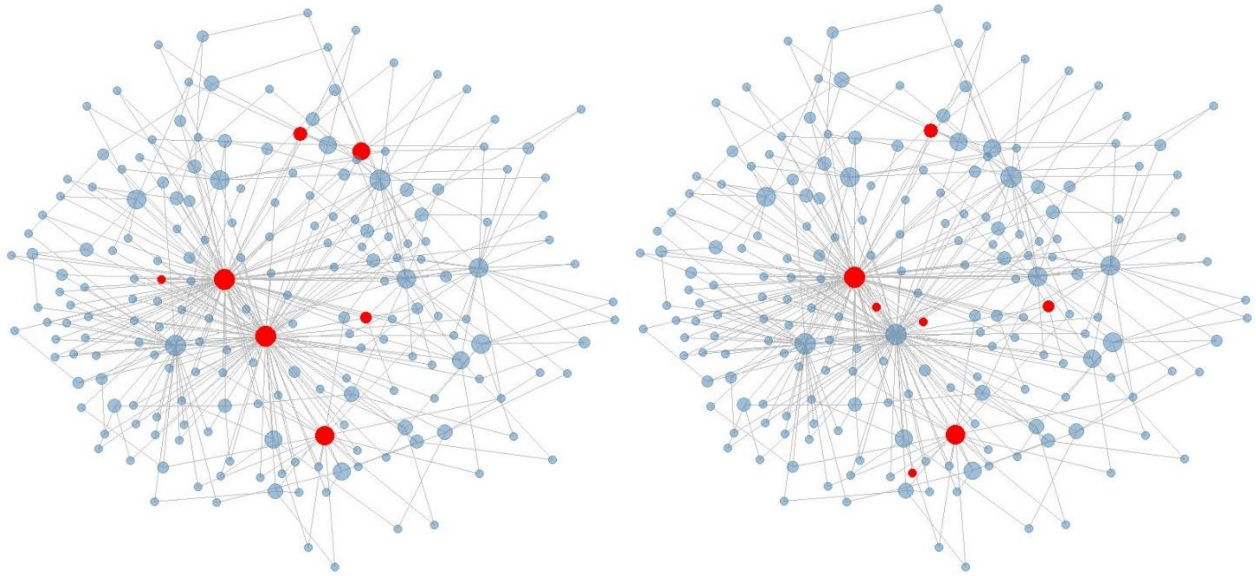
ATTRIBUTI	**VALORI**					**PESO**
Occupazione	Pubblico	Privato	Studente	disoccupato		0.5
Età	20	30	40	50	60	0.3
Sesso	M	F				0.2

I valori degli attributi sono distribuiti sui nodi uniformemente.

Sono stati fatti due test con valore di $\alpha = 1$ nel primo (non considerando quindi la diversità) e $\alpha = 0.3$, nel secondo. I risultati sono i seguenti, restituendo un seed set di 15 elementi:

$\alpha = 1$	$\alpha = 0.3$
spread totale: 16.81510	spread totale: 16.69360
diversity seed set: 14.82913	diversity seed set: 15.45422
seed set ottenuto:	seed set ottenuto:
[0, 1, 2, 15, 12, 6, 9, 7, 65, 36, 35, 83, 54, 74, 77]	[0, 1, 9, 7, 65, 101, 77, 93, 6, 5, 36, 2, 10, 15, 81]

Con il relativo plot del grafo con i seed set ottenuti nei due test (nodi in rosso).

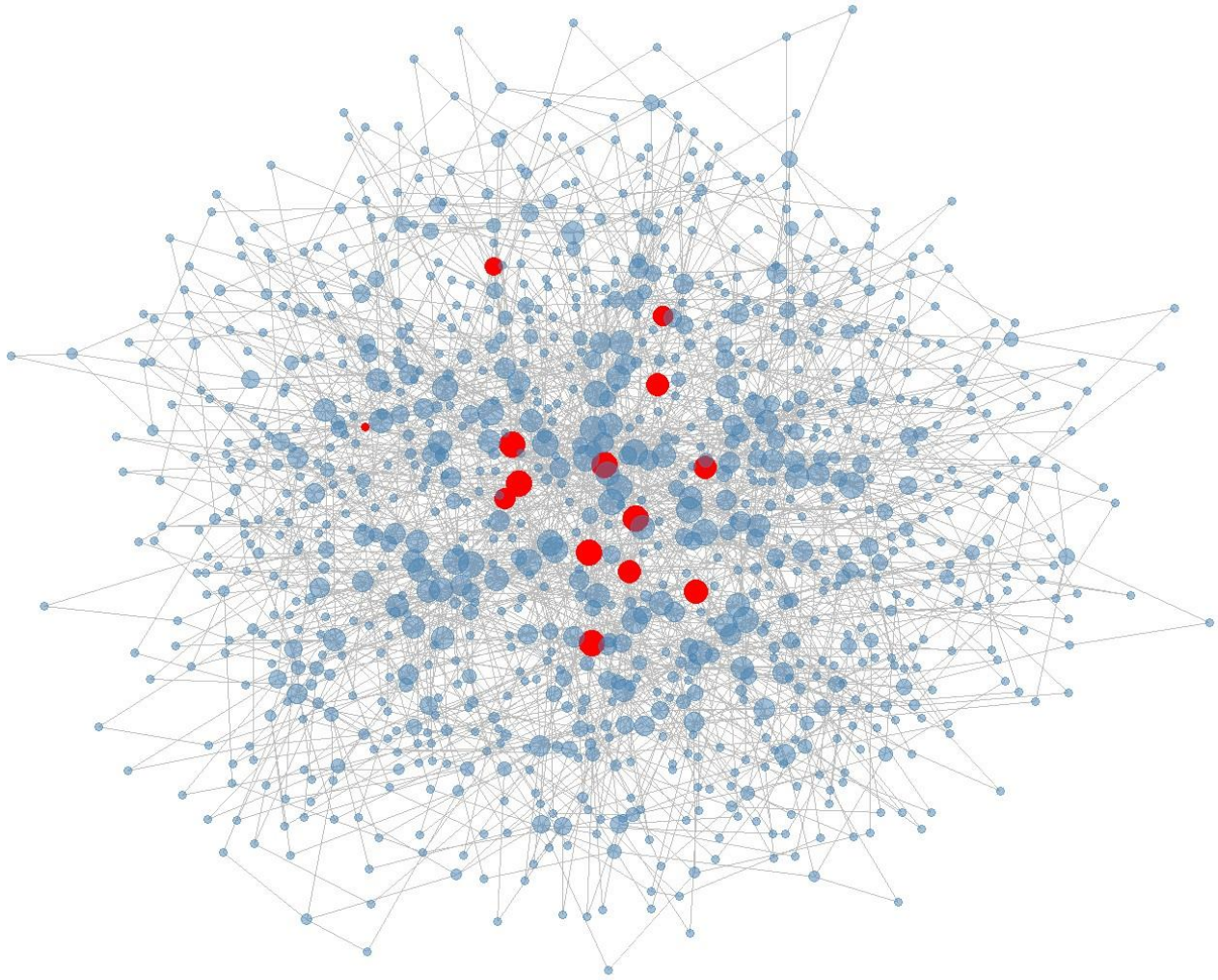


Si può notare che a fronte di un piccolo decremento dello spread c'è un incremento della diversità più marcato.

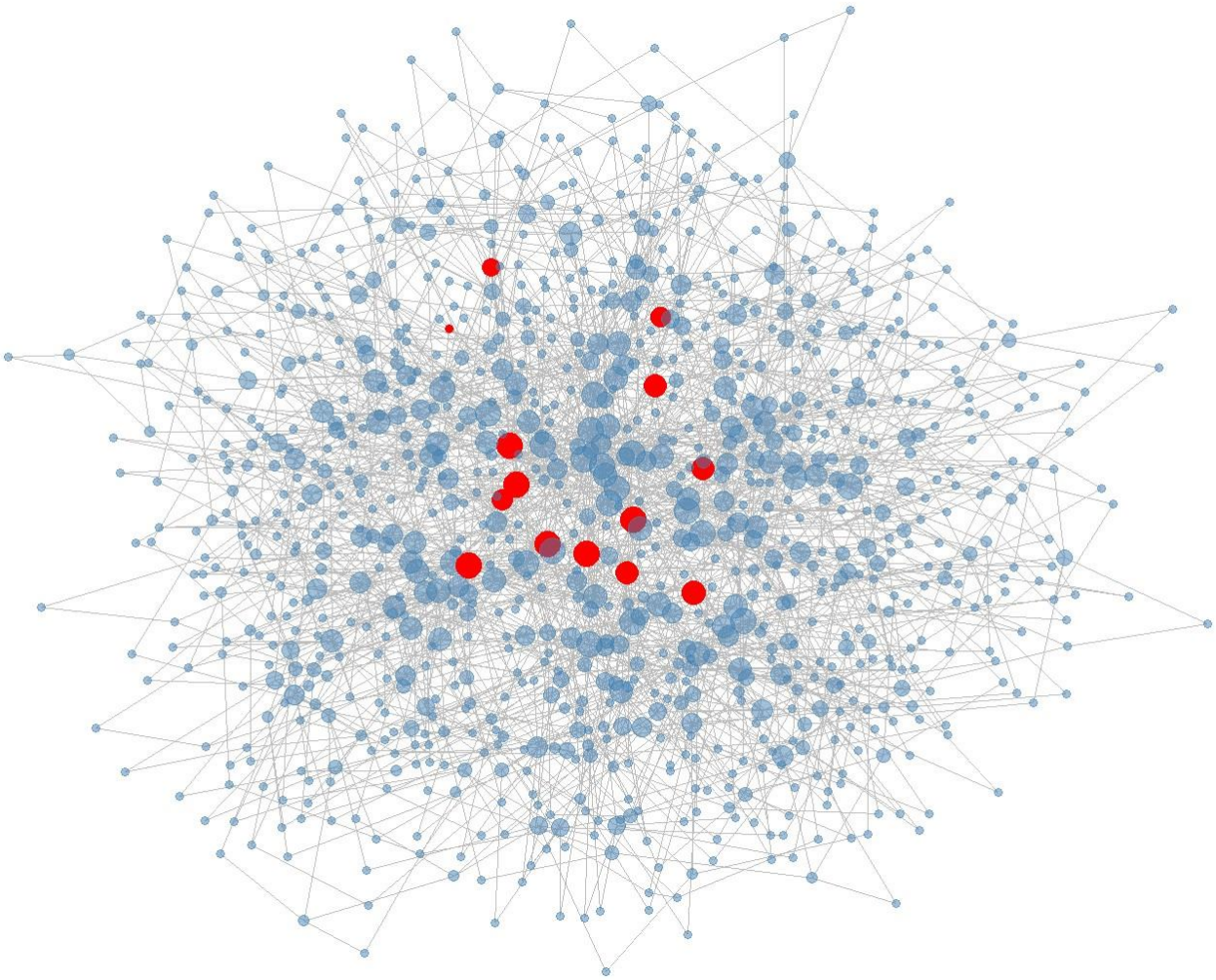
TEST 2. È stato effettuato su un grafo di 1000 nodi e 2000 archi generato anch'esso con il metodo Barabasi-Albert. L'insieme dei valori degli attributi assegnati ai nodi è il medesimo del test precedente con la differenza che i valori sono stati distribuiti secondo una funzione di probabilità esponenziale negativa di parametro $\lambda = 0.7$ (quindi rendendo più presenti i valori di attributo specificati per primi). I risultati sono stati i seguenti.

$\alpha = 1$	$\alpha = 0.1$
spread totale: 196.54064	spread totale: 190.63294
diversity seed set: 12.65941	diversity seed set: 14.73712
seed set ottenuto:	seed set ottenuto:
[0, 1, 10, 18, 4, 3, 8, 12, 16, 11, 15, 62, 43, 48, 67]	[0, 1, 10, 5, 16, 8, 3, 12, 67, 4, 186, 48, 18, 100, 15]

Anche in questo casi la variazione percentuale relativa allo spread è inferiore rispetto a quella della diversità guadagnata. Di seguito il risultato su grafo, nel primo plot si considera $\alpha = 1$:



Nel seguente plot invece si ha $\alpha = 0.1$



TEST 3. Come test finale si è utilizzato un dataset preso sul sito di stanford rinominato "p2p-Gnutella06", che descrive una rete peer-to-peer composta da 8717 nodi e circa 31500 archi. A questa, utilizzando le classi presenti in utilities sono stati aggiunti i pesi sugli archi e ad ogni nodo assegnati valori di attributi, considerando i seguenti domini:

ATTRIBUTI			**VALORI**			**PESO**
Zona	Nord	Centro	Sud			0.5
Sesso	M	F				0.3
Hobby	Calcio	Basket	Rugby	Tennis	Nuoto	0.1
Livello	Pro	Amatoriale				0.1

I valori sono stati distribuiti con una esponenziale negativa di parametro $\lambda = 0.5$. La rete, effettuando il plot con RStudio, è la seguente:

Effettuando due test con valore di α diverso si è ottenuto:

$\alpha = 1$	$\alpha = 0.05$
spread totale: 25.76310	spread totale: 25.31890
diversity seed set: 15.91007	diversity seed set: 17.78364
seed set ottenuto:	seed set ottenuto:
[6494, 6836, 417, 1868, 4471, 5796, 112, 7847, 80, 8246, 916, 3718, 8589, 5483, 424, 465, 2912, 2388, 6513, 909]	[6494, 6836, 417, 5796, 465, 8330, 4471, 7819, 8670, 80, 1868, 424, 112, 1659, 7816, 7847, 3718, 4323, 8246, 916]

I dati confermano quanto detto in precedenza.

Per tutti i test è stata usata la seguente misura di diversità:

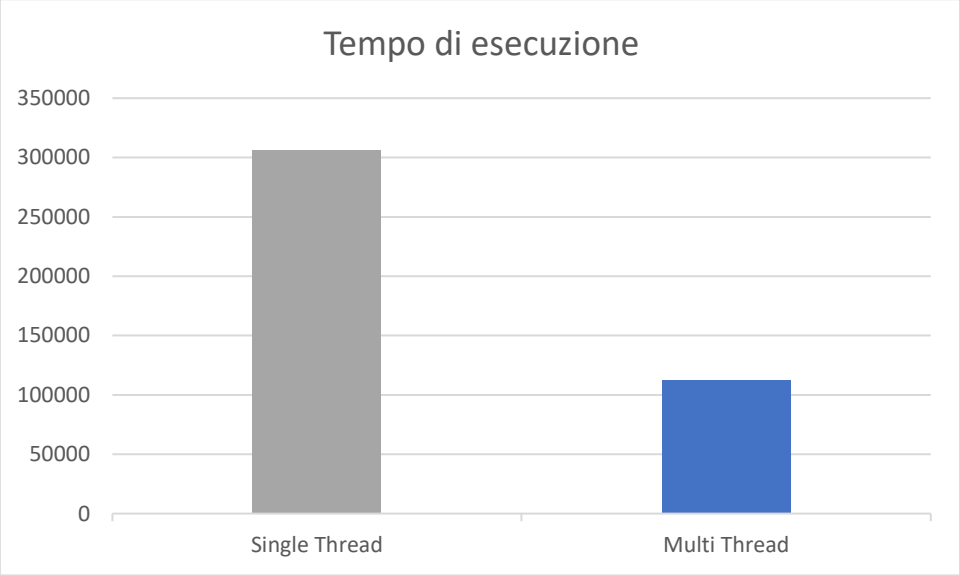
$$div_x(S) = \sum_{v \in V_x} \sum_{i=1}^{num(v)} \frac{1}{i^2}$$

Se si provasse a lanciare l'esecuzione ponendo $\alpha = 0$, si andrebbe a massimizzare solo la diversità senza curarsi dello spread che inevitabilmente crollerebbe. Naturalmente aumentando il numero di valori possibili per ogni attributo aumenterebbe il contributo della diversità, come dualmente, aumentando le dimensioni della rete aumenterebbe l'apporto dello spread allo score di valutazione dei nodi.

Come detto in precedenza una ulteriore ottimizzazione è stata quella di parallelizzare la parte di computazione più onerosa (le chiamate ripetute di SIMPATH-SPREAD, per la valutazione del marginal gain relativo allo spread dei nodi). In particolare, usando un notebook

[caratteristiche pc giovanni]

E lanciando la computazione su un grafo particolarmente ostico (5000 nodi e 50000 archi con alto grado di coesione tra nodi, e valore di $\alpha < 0.1$) si è ottenuto il seguente miglioramento prestazionale.



I valori espressi sono in millisecondi.

Tutti i file di testo relativi ai datasets utilizzati sono in allegato alla relazione e al codice sorgente.

Bibliografia

[1] Simpath: An Efficient Algorithm for Influence Maximization under Linear Threshold Model, Amit Goyal, Wei Lu, Laks V. S. Lakshmanan, University of British Columbia, 2011.

[2] Maximizing the Spread of Influence through a Social Network, David Kempe, Jon Kleinberg, Eva Tardos, Cornell University, Ithaca NY, 2003.

[3] Diversity in Big Data: A Review, Marina Drosou, H.V. Jagadish,Evaggelia Pitoura, and Julia Stoyanovich, University of Ioannina, Ioannina, Greece, 2017.