Reti Sociali e Betweenness Centrality

Gabriele Modena, mat. 108742, e-mail: gabriele.modena@studenti.unitn.it

8 febbraio 2006

Indice

Sommario

Il concetto di betweenness centrality è uno dei cardini dello studio e dell'analisi delle reti sociali (e non). Questa relazione descrive gli aspetti teorici e l'implementazione di un algorimo, elaborato da Ulrik Brandes, che opera con costo $O(n+m)$ per lo spazio e tempo $O(nm)$ e $O(nm + n^2 \log n)$ rispettivamente su grafi non pesati e pesati.

1 Grafi e Reti Sociali

1.1 Rete sociale

Per rete sociale si intende una stuttura determinata dai rapporti che intercorrono tra due o più attori, soggetti o organizzazioni, che ne fanno parte.

Analiticamente possiamo rappresentare una rete sociale con un grafo $G = (V, E)$, i cui vertici V rappresentano gli attori interni alla rete, mentre i lati E rappresentano le relazioni che intercorrono tra loro.

La forma stessa della rete ci fornisce importanti informazioni sull'importanza dei singoli individui che la compongono.

Ai fini dell'analisi di tali strutture è quindi necessario stabilire un indice che consenta di attribuire un punteggio ai singoli attori, in modo da poterne stimare l'importanza.

Uno strumento essenziale per gli analisti è dato dagli indici di centralità di un vertice. Molti di questi si basano sul calcolo dei percorsi minimi che collegano coppie di attori, misurandone la distanza media da altri nodi, oppure la percentuale di percorsi minimi passanti per un vertice

1.2 Caratterizzazione

.

Sia $G = (V, E)$ un grafo. Denotiamo con n il numero dei vertici, m il numero dei lati. Supponiamo per semplicità che il grafo sia *connesso* e *non orientato*, con l'eventuale presenza di cicli o lati multipli. Sia ω una *funzione di peso* sui lati tale che:

- $\omega(e) > 0, \;\; e \in E$ per grafi pesati.
- $\omega(e) = 1, \;\; e \in E$ per grafi non pesati.

Gli indici di centralità che andremo a presentare si basano sul concetto di *cammino minimo*.
Per *cammino dal vertice $s \in V$ al vertice $t \in V$* si intende una successione di lati e vertici che inizia in s e termina in t, tale che ogni lato connette il vertice che lo precede con il successivo.
La *lunghezza* di un cammino è la somma dei pesi dei lati che lo compongono. Denotiamo con $d_G(s,t)$ la *distanza tra il vertice s e il vertice t*, ossia il minimo numero di lati che connettono s a t in G. Segue per definizione che:

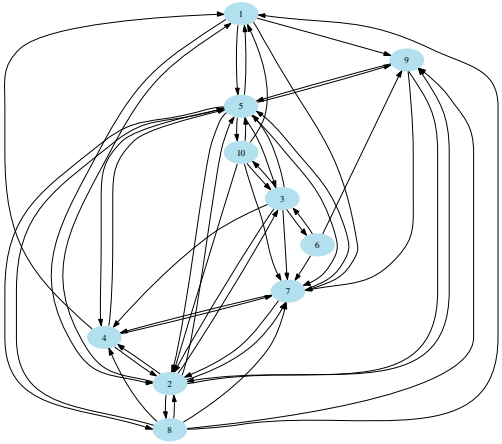
- $d_G(s,s) = 0 \;\;$ per ogni $s \in V$
- $d_G(s,t) = d_G(t,s) \;\;$ per ogni $s, t \in V$

2 Indici di Centralità basati su Cammini Minimi

Sia $\sigma_{st} = \sigma_{ts}$ il numero di camminimi minimi da $s \in V$ a $t \in V$, dove per convezione $\sigma_{ss} = 1$. Sia $\sigma_{st}(v)$ il numero di cammini minimi da s a t passanti per un certo $v \in V$.
Di seguito alcune comuni misure di centralità:

- $C_C(v) = \frac{1}{\sum_{t \in V} d_G(v,t)}$ *closeness centrality* (Sabidussi, 1966)
- $C_G(v) = \frac{1}{max_{t \in V} d_G(v,t)}$ *graph centrality* (Hage and Harart, 1995)
- $C_S(v) = \sum_{s \neq v \neq t \in V} \sigma_{st}(v)$ *stress centrality* (Shimbel, 1953)
- $C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$ *betweenness centrality* (Freeman, 1977; Anthonisse, 1971)

Un alto punteggio di centralità può indicare che un vertice è in grado di raggiungerne altri tramite percorsi brevi, o che un vertice giace su una considerevole parte di percorsi minimi che ne congiungono altri.
Questi indici vengono solitamente normalizzati e fatti giacere nell'intervallo $[0, 1]$. La *non omogeneità* di un indice di centralità viene usata per definire la *centralizzazione* di un grafo rispetto a tale indice. Da un punto di vista computazionale, l'indice di betweenness centrality è quello che viene impiegato più frequentemente nell'analisi delle reti sociali. L'approccio di Freeman vede un attore A essere in posizione favorevole basandosi sul fatto che lo stesso giace sui cammini minimi che collegano altre coppie di attori nel grafo. Piu' gli altri nodi dipendono da A, più A acquista potere all'interno della rete.
Il grafo sottostante rappresenta un esempio della rete di distribuzione globale delle informazioni, proposta dal sociologo [2, D. Knoke] e ripresa nel libro [3, Introduction to social network methods].



Nodo	Indice di CB	Indice di CB ammortizzato
5	17.8333333333333321	0.1238425925925926
2	12.3333333333333339	0.0856481481481482
3	11.6944444444444446	0.0812114197530864
7	2.7500000000000004	0.0190972222222222

9	1.2222222222222221	0.0084876543209877
4	0.8055555555555556	0.0055941358024691
1	0.6666666666666666	0.0046296296296296
10	0.3611111111111111	0.0025077160493827
6	0.3333333333333333	0.0023148148148148
8	0.0000000000000000	0.0000000000000000

L'algoritmo elaborato da Ulrik Brandes, oltre ad un notevole miglioramento in termini di costo computazionale, ha il vantaggio di poter essere adattato al calcolo di altri indici di centralità virtualmente senza costi aggiuntivi.

2.1 Criterio di Bellman

Nel calcolo di C_B, è importante ricordare il seguente
lemma:
Un vertice $v \in V$ giace su un cammino minimo tra i vertici $s, t \in V$, se e solo se $d_G(s,t) = d_G(s,v) + d_G(v,t)$

Date la distanza a coppie e il conteggio di percorsi minimi, la *dipendenza di coppia* $\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$, il numero di percorsi minimi tra s e t su cui giace v è dato da:

$$\sigma_{st}(v) = \begin{cases} 0 & d_G(s,t) < d_G(s,v) + d_G(v,t) \\ \sigma_{sv}\sigma_{vt} & d_G(s,t) \geq d_G(s,v) + d_G(v,t) \end{cases}$$

Il calcolo di C_B rispetto ad un vertice v si riduce alla somma delle dipendenze di coppia di tutte le coppie su quel vertice

$$C_B = \sum_{s \neq v \neq t \in V} \delta_{st}(v)$$

C_B viene determinato in due fasi:

1. si calcola la lunghezza e il numero dei camminimi minimi tra tutte le coppie
2. si sommano tutte le dipendenze di coppia

3 Betweenness Centrality

Osserviamo che, ad una prima analisi, la complessità nel determinare C_B è dominata dalla seconda fase che richiede tempo $\Theta(n^3)$ per la somma e spazio $\Theta(n^2)$ per salvare le dipendenze di coppia. La soluzione proposta da Ulrik Brandes pone rimedio a questa situazione, comportando costi meno onerosi sia in termini di spazio che di tempo.

3.1 Conteggio dei cammini minimi

Lemma: Conteggio algebrico dei cammini: *Sia* $A^k = (a^k_{uv})_{u,v \in V}$ *la k-esima potenza nella matrice d'adiacenza di un grafo non pesato. Allora* a^k_{uv} *è uguale al numero di percorsi da u a v di lunghezza esattamente k.* Dato che e' necessario sommare $\Theta(n^2)$ dipendenze di coppia per ogni vertice, il costo di esecuzione complessivo è dominato dal tempo impiegato ad effettuare la moltiplicazione tra matrici. Questo approccio comporta più calcoli del dovuto, in quanto quello che ci interessa è il numero di percorsi minimi tra ogni coppia di vertici. Sfruttando la sparsità di istanze tipiche, si possono contare i percorsi minimi usando algoritmi di attraversamento.

Definiamo ora l'insieme dei *predecessori* di un vertice v sui percorsi minimi che partono da s come:

$$P_S(v) = \{u \in V : \{u, v\} \in E, d_G(s, v) = d_G(s, u) + \omega(u, v)\}$$

Lemma (Conteggio combinatorio dei cammini minimi):
Per $s \neq v \in V$

$$\sigma_{sv} = \sum_{u \in P_s(v)} \sigma_{su}$$

Dimostrazione:
Poichè tutti i pesi dei lati sono positivi, l'ultimo lato di ogni percorso minimo da s a v è un lato $u, v \in E$ tale che $d_G(s, u) < d_G(s, v)$. Ne segue che il numero di percorsi minimi da s a v che terminano con questo lato è pari al numero di percorsi minimi da s a u. Ciò è garantito dal *Criterio di Bellman*. QED.
. Grazie a questo lemma è possibile utilizzare sia *Diijkstra* che una visita *BFS* per contare i cammini minimi. BFS richiede un tempo $O(m)$, mentre Dijkstra esegue in tempo $O(m + n \log n)$ se la coda di priorità è implementata con un heap di Fibonacci.

Corollario: *Data una sorgente* $s \in V$*, sia la lunghezza che il numero di percorsi minimi verso altri vertici possono essere determinati in tempo* $O(m + n \log m)$ *per grafi pesati e in tempo* $O(m)$ *per grafi non pesati.*

Ne consegue che $\sigma_{st} \in V$ può essere calcolato in tempo $O(nm)$ per grafi non pesati e in tempo $O(nm + n^2 \log n)$ per grafi pesati.

Il corollario implica che il costo computazionale dell'algoritmo è dominato dal tempo $\Theta(n^3)$ necessario a sommare le dipendenze di coppia. È possibile diminuire tale costo in maniera sostanziale, tramite l'accumulazione delle somme parziali delle dipendenze di coppia.

3.2 Accumulazione delle dipendenze di coppia

Per prima cosa definiamo la *dipendenza* di un vertice $s \in V$ da un vertive $v \in V$ come

$$\delta_{s*}(v) = \sum_{t \in V} \delta_{st}(v)$$

È importante osservare che queste somme parziali rispondono ad una relazione ricorsiva.
Lemma: *Se esiste un unico percorso minimo da* $s \in V$ *ad ogni* $t \in V$*, la dipendenza di s da ogni* $v \in V$*, e' data da*

$$\delta_{s*}(v) = \sum_{w:v \in P_s(w)} (1 + \delta_{s*}(w))$$

Dimostrazione:
L'assunzione implica che i vertici e i lati di tutti i percorsi minimi formino un albero. Quindi abbiamo che v giace su tutti i possibili percorsi tra $s, t \in V$ o su nessuno, da cui $\delta_{st}(v)$ e uguale o a 1 o a 0. Inoltre, v giace su tutti quei percorsi minimi dei vertici di cui è un predecessore e su tutti i percorsi minimi su cui questi ultimi giacciono.

Teorma: *La dipendenza di $s \in V$ da un vertice $v \in V$, è data da*

$$\delta_{s*}(v) = \sum_{w:v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} (1 + \delta_{s*}(w))$$

Dimostrazione:
Ricordiamo che $\delta_{st}(v) > 0$ solo per i vertici $t \in V$ t.c. v giace su almeno un percorso minimo da s a t e notiamo che su ognuno di questi percorsi c'è un unico lato u, w con $v \in P[w]$. Estendiamo la definizione dipendenza di coppia per includere un lato $e \in E$, definendo

$$\delta_{st}(v, e) = \frac{\sigma_{st}(v,e)}{\sigma_{st}}$$

dove σ_{st} è il numero di percorsi minimi da s a t che contengono sia v che e. Quindi:

$$\delta_{s*}(v) = \sum_{t \in V} \delta_{st}(v) = \sum_{t \in V} \sum_{w:v \in P_s(w)} \delta_{st}(v, \{v, w\}) = \sum_{w:v \in P_s(w)} \sum_{t \in V} \delta_{st}(v, \{v, w\}).$$

Sia w un vertice t.c. $v \in P_s(w)$. Dei percorsi minimi da s a w σ_{sw}, σ_{sv} vanno prima da s in v e poi usano $\{v, w\}$. Di conseguenza $\frac{\sigma_{sv}}{\sigma_{sw}} \sigma_{st}(v)$ percorsi minimi da s a un certo $t \neq w$ contengono v e $\{v, w\}$. Ne segue che la dipendenza di coppia di s e t da v e $\{v, w\}$ è

$$\delta_{st}(v, \{v, w\}) = \begin{cases} \frac{\sigma_{sv}}{\sigma_{sw}} & t = w \\ \frac{\sigma_{sv}}{\sigma_{vt}} \frac{\sigma_{st}(w)}{\sigma_{st}} & t \neq w \end{cases}$$

Inserendo questo risultato nell'equazione precedente arriviamo a:

$$\sum_{w:v \in P_s(w)} \sum_{t \in V} \delta_{st}(v, \{v, w\}) = \sum_{w:v \in P_s(w)} \left(\frac{\sigma_{sv}}{\sigma_{sw}} + \sum_{t \in V - \{w\}} \frac{\sigma_{sv}}{\sigma_{sw}} \frac{\sigma_{st}(w)}{\sigma_{st}}\right) \sum_{w:v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} (1 + \delta_{s*}(w)).$$

QED

Corollario: *Dato il grafo orientato aciclico dei cammini minimi da $s \in V$ in G, le dipendenze di s da ogni altro vertice possono essere calcolate in tempo $O(m)$ e spazio $O(n + m)$.*
Dimostrazione:
Il procedimento è il seguente: si attraversano i vertici in ordine non crescente della loro distanza da s e si sommano le dipendenze sfruttando il teorema precedente. È necessario salvare una dipendenza per vertice e delle liste di padri. C'è al più un elemento per lato in ognuna di queste liste. QED

Da questo corollario deriva il fatto che si possa calcolare l'indice di betweenness centrality risolvendo un singolo problema di percorso minimo per ogni vertice. Al termine di ogni iterazione, le dipendenze della sorgente da un vertice vengono sommate al punteggio di centralità di quel vertice.

Teorema: *L'indice di betweenness centrality può essere calcolato in tempo* $O(nm + n^2 \log n)$ *e in spazio* $O(n + m)$ *per grafi pesati. Per grafi non pesati, il tempo di esecuzione si riduce a* $O(nm)$.
Dimostrazione:
Segue dalle considerazioni precedenti.

4 Algoritmo di Ulrik Brandes per il calcolo di C_B

Betweenness centrality nei grafi non pesati.

$C_B[v] = 0, \forall v \in V$;
for $s \in V$ **do**
    S = stack vuoto
    $P[w]$ = lista vuota, $\forall w \in V$;
    $\sigma[t] = 0, \forall t \in V$; $\sigma[s] = 1$
    $d[t] = -1, \forall t \in V$; $d[s] = 0$
    Q = coda vuota;
    push (s,Q);
    while Q *non vuoto* **do**
        v = pop(Q);
        push(v,S);
        foreach w *vicino di* v **do**
            /* Incontro w per la prima volta */
            if $d[w]0$ **then**
                push(w, Q);
                $d[w] = d[v] + 1$;
            endif
            /* Percorso minimo da w passante per v? */
            if $d[w] = d[v] + 1$ **then**
                $\sigma[w] = \sigma[w] + \sigma[v]$;
                append($v, P[w]$);
            endif
        endforeach
    endwhile
    $\delta[v] = 0, \forall v \in V$;
/* S contiene i vertici in ordine di distanza non crescente dalla sorgente s*/
    while S non è vuoto **do**
        w = pop(S);
        for $s \in P[w]$ **do** $\delta[v] = \delta[v] + \frac{\sigma[v]}{\sigma[w]}(1 + \delta[w])$;
    if $w \neq s$ **then** $C_B[w] = C_B[w] + \delta[w]$;
endfor

4.1 Descrizione

Il codice proposto da Ulrik Brandes per risolvere il problema di C_B nel caso di grafi non pesati è una rivisitazione dell'algoritmo di BFS, che però ottimizza sia la procedura di conteggio dei percorsi minimi sia la somma dei risultati parziali ottenuti. L'algoritmo itera ogni vertice $s \in V$, calcolando le dipendenze e il numero di percorsi minimi che lo collegano ad ogni altro nodo $w \in V$. Per prima cosa vengono create le strutture dati necessarie a salvare i risultati dell'operazione. Si inizializzano quindi $\forall v \in V$ i valori di σ e d, numero di percorsi minimi su cui il vertice giace e distanza dall'origine, rispetto al vertice s su cui stiamo iterando. A questo punto salviamo s sulla coda dei vertici visitati. Nel ciclo while avviene la fase di visita vera e propria. Per prima cosa si estrae un vertice v dalla coda e lo si salva sullo stack S, che al termine del ciclo S conterrà i vertici del grafo ordinati in modo non crescente rispetto alla sorgente s. Si procede analizzando tutti i vertici w vicini di v. $\forall\ w$ controlliamo se lo stiamo visitando per la prima volta, nel qual caso la sua distanza da s viene incrementata di $1 + d[v]$ e il vertice viene posto sulla coda dei nodi scoperti. La proprietà del *conteggio combinatorio* ci consente di determinare se stiamo raggiungendo w tramite v passando per un cammino minimo, ossia se v è padre di w. In caso affermativo, incrementiamo il contatore del numero di percorsi minimi passanti per w e inseriamo v nella lista $P[w]$ dei padri di w. Queste operazioni vengono ripetute per tutti i vicini del nodo v.

Una volta visitati tutti i nodi, si procede al calcolo del valore di C_B come somma di dipendenza di coppia. La correttezza e il costo computazionale di questa operazione ci sono garantite dal teorema visto in precedenza.

Una volta calcolato il C_B di ogni nodo $v \in V$ rispetto alla sorgente s, si procede ripetendo l'algoritmo per il successore di s nel grafo.

5 Implementazione

Questa sezione contiene la descrizione dell'implementazione da me realizzata per il calcolo di C_B su grafi non pesati, orientati e non orientati. Il codice sorgente è così organizzato:

- *graph.c, graph.h*: header e implementazione della struttura dati usata per gestire i grafi.
- *stack.c, stack.h*: header e implementazione delle strutture dati pila e coda.
- *graphio.c, graphio.h*: funzioni che gestiscono input (lettura da file) output (esportazione in formato *dot*) per il caricamento e il salvataggio di grafi.
- *centrality.c, centrality.h*: header e implementazione dell'algoritmo di Ulrik Brandes per il calcolo di C_B su grafi *non pesati*.

5.1 Rappresentazione di un grafo: *graph.h*

Ho adottato l'implementazione come lista concatenata.

5.1.1 Vertici

```
typedef struct tvertex
{
  v_tag id;

  struct tedge *edges;
  struct tvertex *next;

  int visited;
  struct cbData *data;

} t_vertex;
```

Ogni nodo del grafo è descritto da:

- un id numerico intero positivo, *v_tag*.
- un puntatore *next* al vertice che segue nella lista.
- un puntatore *edges* che tiene traccia dei lati uscenti.
- una variabile booleana *visited* che tiene traccia dello stato del lato in eventuali algoritmi di visita.
- un puntatatore *data* ad eventuali dati aggiuntivi. Qui *data* conterràinformazioni relative al calcolo di C_B

5.1.2 Lati

```
typedef struct tedge
{
  int weight;
  struct tvertex *links;
  struct tedge *next;
} t_edge;
```

I lati uscenti da un nodo sono rappresentati tramite una lista concatenata, così strutturata:

- un intero *weight* che ne rappresenta il peso.
- un puntatore *links* al vertice verso cui il lato è diretto.
- un puntatore *next* al lato successivo nella lista.

5.1.3 Grafo

```
typedef struct tgraph
{
  int num_vertices;
  int is_directed;
  struct tvertex *vertices;
} t_graph;
```

A questo punto è possibile rappresentare l'intero grafo tramite:

- un intero *num_vertices* che tiene conto del numero di vertici.
- una variabile boolena *is_directed* impostata a $TRUE$ se il grafo è orientato e a $FALSE$ se il grafo è non orientato.
- un puntatore *vertices* al vertice che da origine al grafo (il primo in ordine di inserimento).

5.2 Operazioni sul grafo

graph.h fornisce inoltre le seguenti funzioni:

5.2.1 Crea un nuovo grafo

```
t_graph *createGraph(int is_direct)
```

Alloca memoria per un nuovo grafo e ritorna un puntatore.
is_direct è una variabile booleana impostata a $TRUE$ se il grafo creato è orientato, $FALSE$ altrimenti.

5.2.2 Distruggi un grafo

```
void destroyGraph(t_graph *graph)
```

Dealloca la memoria riservata al grafo *graph*.

5.2.3 Aggiungere un vertice

```
t_vertex *addVertex(t_graph *graph, v_tag id)
```

Costo: $O(n)$
Aggiunge un vertice *id* al grafo *graph* e ne ritorna un puntatore.

5.2.4 Rimuovere un vertice

```
void removeVertex(t_graph *graph, t_vertex *v)
```

Costo: $O(nm)$
Rimuove un nodo e tutti gli archi a lui associati
Aggiungere un lato

```
int addEdge(t_graph *graph, t_vertex *srcVertex, t_vertex *dstVertex, int weight)
```

Costo: $O(n + m)$
Inserisce un nuovo lato con peso *weight* che congiunge il vertice *srcVertex* al vertice *dstVertex*. La funzione ritorna *-1* in caso di errore, *0* nel caso in cui l'operazione sia stata eseguita con successo.

5.2.5 Rimuovere un lato

```
void removeEdge(t_graph *graph, t_vertex *src, t_vertex *dst)
```

Costo: $O(n + m)$
Rimuove un lato e aggiorna la lista dei puntatori del vertice sorgente *src*.

5.2.6 Verificare la presenza di un vertice

```
int isInGraph(t_graph *graph, v_tag id)
```

Costo: $O(n)$
Controlla se il vertice *id* fa parte del grafo *graph*. In caso affermativo ritorna *0*, altrimenti ritorna *-1*

5.3 Funzioni di I/O: *graphio.h*

5.3.1 Caricamento

È possibile caricare un grafo da un file di testo (lista di lati) in formato csv, i cui campi sono separati da spazi.

```
"Vertice di partenza" "Vertice di arrivo" "Peso del lato"
```

L'operazione di caricamento avviene tramite la funzione

```
int readFromCSV(t_graph *graph, char *filename)
```

che ha come argomenti un puntatore *graph* ad un grafo e il percorso *filename* del file da caricare.
La funzione ritorna *-1* in caso di errore, *0* nel caso in cui l'operazione sia stata eseguita con successo.

5.3.2 Stampa del grafo

```
void dumpGraph(t_graph *graph)
```

Stampa a schermo una rappresentazione del grafo *graph* come lista concatenata.

5.3.3 Stampa del grafo con sintassi *dot*

```
void dumpDotGraph(t_graph *graph)
```

Stampa a schermo una rappresentazione del grafo *graph* con la sintassi *dot* del pacchetto *graphviz*.

5.4 Betweenness Centrality su grafi non pesati: *centrality.h*

5.5 Struttura

Ad ogni nodo del grafo è associata la struttura dati *cb_data* composta da:

```
typedef struct cbData {
  double sp_num;
  double distance;
  double dependency;
  struct tlParents *parents;
  double rank;
} cb_data;
```

- un contatore *sp_num* che tiene conto dei precorsi minimi passanti per il nodo.
- un contatore *distance* che tiene conto della distanza dal vertice sorgente.
- il valore *dependenxy* delle dipendenze $\delta(v)$ del nodo.
- una lista concatenata *parents* dei padri $P(v)$ del nodo.
- il valore *rank* dell'indice di C_B non normalizzato.

5.5.1 Calcolo di C_B

Costo: tempo $O(nm)$, spazio $O(n+m)$

```
void betweennessCentrality(t_graph *graph)
```

Calcola il valore di centralità dei vertici che compogono la rete *graph*.

5.5.2 Stampa i valori di C_B

```
void printGraphCB(t_graph *graph)
```

Stampa a schermo i valori di C_B dei singoli vertici. La prima colonna indica l'id di un nodo, la seconda il suo valore di $C_B(v)$, la terza il valore *ammortizzato* di $C_B(v)$ secondo [5, WikiPedia], la quarta il valore calcolato dal sofware Pajek (che ammortizza dividendo C_B per $(n-1)(n-2)$). Si noti che in caso di grafo non orientato Pajek non divide il risultato ottenuto a metà generando un output diverso da quello della mia implementazione.

Il valore ammortizzato di un nodo v ́dato da: $\frac{\frac{C_B(v)}{(n-1)(n-2)}}{2}$.

5.6 Osservazioni

La cartella *src* contiene i sorgenti del programma, *doc* contiene la relazione in formato pdf e il sorgente . *set* contiene grafi e dataset di cui ho calcolato C_B, *graph* contiene il grafo di Knoke in formato *dot*.

Per compilare il programma è possibile eseguire il comando *make cb* nella directory *centrality*, una volta decompresso l'archivio tbz2.

```
./cb -h

Utilizzo:
-b : esegue il calcolo di cb e ne stampa a schermo i risulati
-f : carica file
-o : grafo orientato
-d : genera l'output del grafo in formato dot

Esempio:
cb -o -b -f net.csv | sort.sh
cb -o -d -f net.csv
```

Lo script *sort.h* ordina in maniera decrescente l'output prodotto *cb*, prendendo come argomento il flusso passato via pipe o un file precedentemente salvato. Utilizzando il pacchetto [6, graphviz] è possibile generare un'immagine che rappresenta il grafo dell'output ottenuto usando l'opzione $-d$.

```
dot -Tps net.dot -o net.ps
```

Il programma è stato testato (compilato ed eseguito con esito positivo) sulle seguenti piattaforme:

- Linux x86 con gcc3.4
- Linux x86 con gcc4.0
- Linux ppc con gcc3.4
- FreeBSD-6.0 x86 con gcc3.4

La correttezza dei risultati è stata verificata eseguendo dei test e controlli incrociato con il tool [7][pajek] citato dall'autore del paper.

6 Applicazioni di C_B ed esempi di reti sociali (e non)

In rete si possono trovare numerosi esempi e dataset di ricerche sull'interazione di attori in una rete [4, TrustMetricsWiki]. Le applicazioni di questo modello matematico spaziano dalla chimica (interazione di proteine) al cinema (dataset di imdb). Questa sezione raccoglie e descrive brevemente alcuni di questi esempi. Dove possibile, ossia quando calcolabile in tempo macchina ragionevole secondo i mezzi che ho a disposizione, ho eseguito un test di C_B con l'algoritmo qui implementato.

6.1 Diametro del World Wide Web

```
Fonte: http://www.nd.edu/%7Enetworks/database/Papers/science.pdf
Dataset: http://www.nd.edu/~networks/resources/www/www.dat.gz
Formato: Da -> A (arco diretto).
```

Il dataset è relativo ad un articolo apparso sul numero 401 di Nature (1999). Gli autori hanno sondato il www con un agente che ha registrato il grado di connettività locale e salvato in un database la topologia della rete. L'agente ha ricorsivamente proceduto andando a recuperare i documenti a loro volta citati. Ogni numero rappresenta una pagina web all'interno del dominio *nd.edu*.

6.2 Rete sociale degli attori

```
Fonte: http://www.nd.edu/%7Enetworks/database/Papers/science.pdf
```

Articolo apparso sul numero 286 di Sicence (1999). Il dataset è relativo alle relazioni tra attori che hanno recitato negli stessi film, secondo database di imdb.

6.3 Percorsi metabolici

```
Fonte: http://www.cosin.org/extra/data/metabolic/
Dataset: http://www.cosin.org/extra/data/metabolic/ecoli.metabolic.links
Formato: Da -> A (arco diretto).
```

Database dei percosri metabolici derivato da Kyoto Encyclopedia of Genes and Genomes (KEGG). I nodi sono sostanze prodotte dal metabolismo. Un arco è posizionato tra due nodi se gli stessi sono coinvolti nella stessa reazione catalitica.

```
cb -v 895 -o -b set/ecoli/meta/ecoli.metabolic.links
```

6.4 DIMES

```
Fonte: http://www.netdimes.org
Dataset: http://www.netdimes.org/PublicData.html
Formato Da -> A (orientato per BGP, non orientato per DIMES)
```

Il progetto DIMES rilascia pubblicamente su base mensile i grafi dei Sistemi Autonomi e dei router della rete Internet. I dataset sono divisi in:

- DIMESEdges - insieme dei lati di un AS trovati nel mese e visitati almeno due volte.
- BGPEdges - insieme di livelli AS affetti da aggiornamenti di BGP, rilevati dal progetto routeviews.

Dato che questi dataset hanno un formato leggermente diverso da quello da me adottato, *set/dimes* contiene lo script di conversione *dimes2me.sh*.

6.5 PGP Web of Trust

```
Fonte: http://bcn.boulder.co.us/~neal/pgpstat/
```

Analisi di keyring pgp mirata a stabilire il rapporto di fiducia di un attore, in base alla quantità di firme ricevute.

Riferimenti bibliografici

[1] **Ulrik Brandes, 2001**. *A Faster Algorithm for Betweenness Centrality*. Journal of Mathematical Sociology 25(2):163-17

[2] **D. Knoke**. *Personal web page*. http://www.soc.umn.edu/faculty/Knoke.htm

[3] **Hanneman, Riddle, 2005**. *Introduction to social network methods*. Riverside, CA: University of California, Riverside. Published in digital form at http://faculty.ucr.edu/ hanneman/

[4] **TrustMetricsWiki**. *Available Datasets*. http://moloko.itc.it/trustmetricswiki/moin.cgi/AvailableDatasets

[5] **WikiPedia**. *Betwenness Centrality.*
http://en.wikipedia.org/wiki/Betweenness_centrality

[6] **Graphviz**. *Graphviz - Graph Visualization Software.*
http://www.graphviz.org/

[7] **Pajek**.*http://vlado.fmf.uni-lj.si/pub/networks/pajek/*
http://vlado.fmf.uni-lj.si/pub/networks/pajek/