
Università di Catania
Facoltà di SS. MM. FF. NN.
Dipartimento di Matematica e Informatica
Corso di Laurea in Informatica di primo livello

Relazione sull'attività di stage interno

Studente: Fabio Rinnone (matricola 667/001087)

Tutor universitario: prof.ssa Rosalba Giugno

Tutor esterno: dott. Giuseppe Pigola

Anno accademico 2009/10

1. Premessa

L'attività di stage interno è consistita nell'implementazione di un nuovo algoritmo per il calcolo, a partire da una query fornita in input dall'utente, dei match totali e distinti trovati su strutture dati di tipo grafo. L'algoritmo è stato implementato su Netmatch, plugin realizzato in linguaggio Java per l'applicazione Cytoscape.

Cytoscape (<http://www.cytoscape.org>) è un programma rilasciato sotto licenza GPL utilizzato nel settore della bioinformatica per la visualizzazione delle reti d'interazione molecolare. Molte caratteristiche aggiuntive sono rese disponibili mediante una serie di plugin. Vi sono, infatti, plugin per la profilatura delle reti molecolari, per nuovi layout, per il supporto di file aggiuntivi e per la connessione con appositi database. Tuttavia Cytoscape può essere anche utilizzato per creare, visualizzare ed analizzare grafi di qualsiasi tipo.

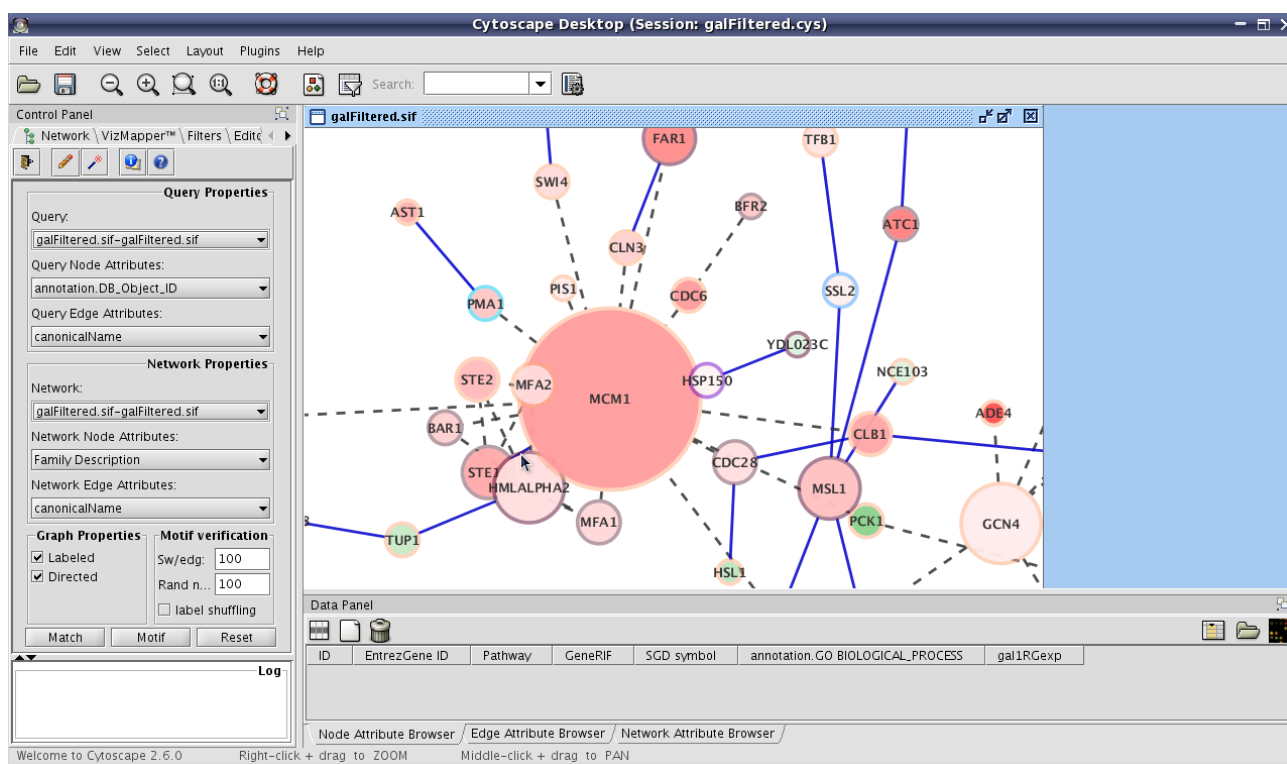


Illustrazione 1: Cytoscape in esecuzione. Il pannello a sinistra è Netmatch

Netmatch è un plugin per Cytoscape che permette all'utente di creare delle query che possono essere passate ad uno o più grafi caricati nello spazio di lavoro di Cytoscape ([1]). Una query in Netmatch è un grafo: esso può essere o uno stesso grafo nello spazio di lavoro di Cytoscape o può essere generato tramite un wizard o, come ulteriore alternativa, creato appositamente mediante un'interfaccia che permette la rappresentazione grafica di nodi ed archi del grafo. Il risultato della query sarà un insieme di sottografi del grafo originale, di cui, per ogni singolo sottografo, verranno rappresentati i valori degli attributi e, se richiesta, anche una loro

rappresentazione grafica. Il plugin si occupa anche di mostrare il numero totale di match nel grafo trovati a partire da una query ed il numero di match distinti. Il motivo di questa distinzione sta nel fatto che, data una query di partenza, possono essere restituiti in output sottografi equivalenti (che rappresentano quindi, in buona sostanza, la stessa informazione), ma che sono stati trovati in fasi diverse dell'esecuzione dell'algoritmo e che hanno come vertice iniziale, di volta in volta, un vertice diverso del sottografo rappresentante la query. Volendo fare un esempio banale, se si vuole ricercare in una rete ogni ciclo costituito da tre vertici, e supponendo che nella rete ne sia presente uno solo, troveremo come risultato tre rappresentazioni diverse, ognuna avente come vertice iniziale uno dei tre nodi della query. Tuttavia, appare evidente che, in questo banale esempio, i tre risultati rappresentano la medesima informazione: è necessario dunque eliminare le informazioni ridondanti e superflue ed è sufficiente mostrare in output un singolo match.

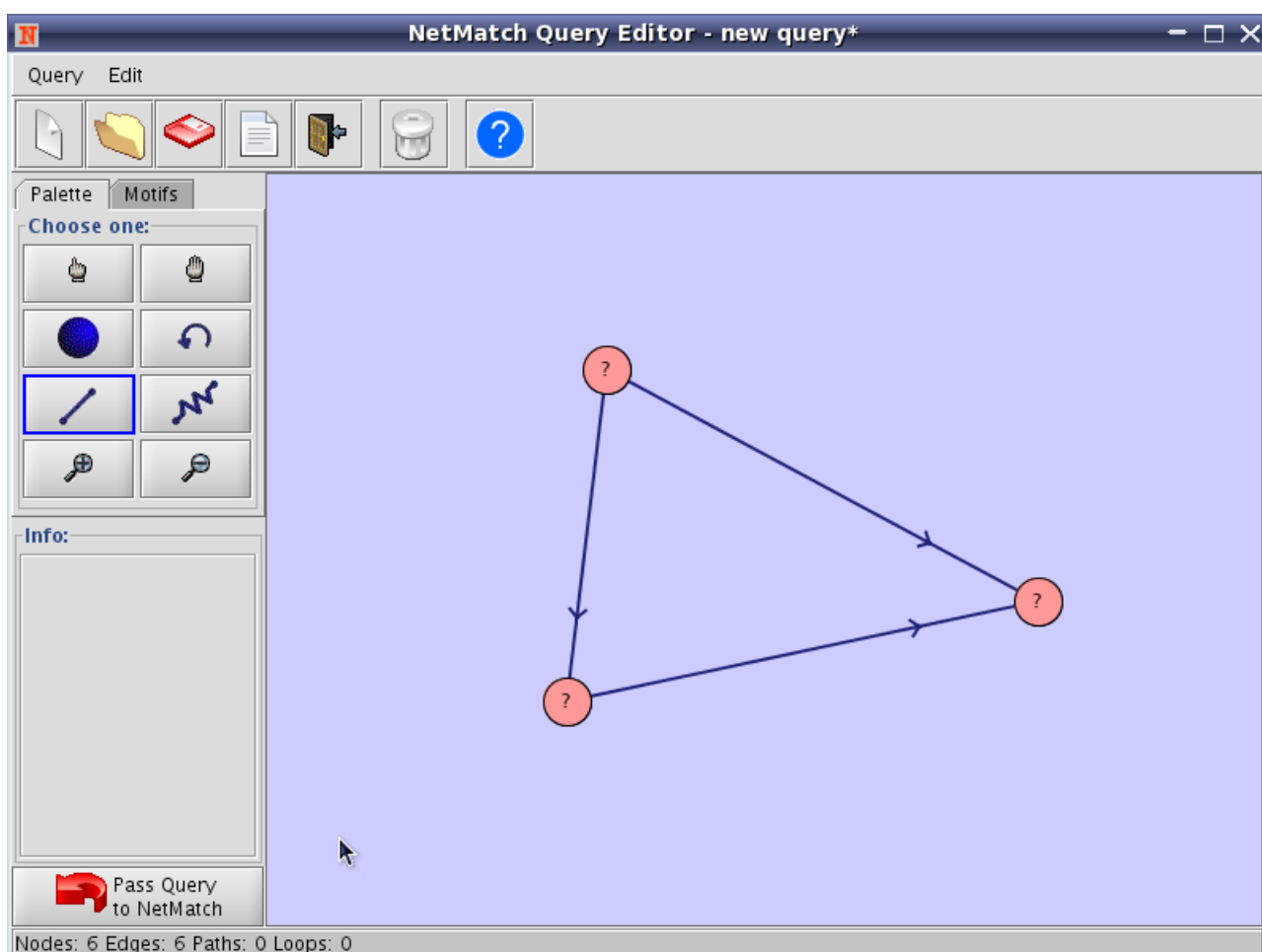


Illustrazione 2: Una query creata con l'editor grafico

Il numero totale di match trovato è dato da tutti i match (comprese le occorrenze duplicate e ridondanti), il numero di match distinti è dato, invece, da tutti i match escludendo le occorrenze duplicate.

La versione 2.1 di Netmatch utilizzava un algoritmo poco efficiente per l'eliminazione delle occorrenze

duplicate. L'attività di stage interno è consistita, quindi, nella realizzazione di un nuovo algoritmo che abbattesse radicalmente i tempi di esecuzione della ricerca dei match distinti.

2. Implementazione

Cytoscape utilizza una sua struttura dati per la rappresentazione dei grafi. La classe Java usata per la rappresentazione dei grafi si chiama `CyNetwork`; i nodi sono rappresentati mediante la classe `CyNode` e gli archi mediante un'apposita classe `CyEdge`. In Netmatch la rappresentazione della query e del target (ossia il grafo su cui applicare la query) avviene mediante la struttura dati `ArrayList` disponibile nel package `java.util` delle API di Java. La classe che gestisce tale struttura è `ArrayList` ed è un'implementazione dell'interfaccia `List` che permette, quindi, tutte le operazioni disponibili in una `List` ed in più consente di manipolare la dimensione dell'array utilizzato internamente per la rappresentazione della struttura dati ([2]). Ogni elemento dell'`ArrayList` rappresentante la query ed il target è rappresentato sotto forma di array di interi, di cui, ogni elemento, è un identificatore di un nodo. Il plugin Netmatch, per prima cosa converte le strutture dal formato utilizzato da Cytoscape (`CyNetwork`) al formato di `ArrayList`. Eseguito l'algoritmo e trovati i match (quindi popolato l'`ArrayList` rappresentante il target) le strutture vengono nuovamente tradotte nel formato di Cytoscape.

L'implementazione originale dell'algoritmo prevedeva la costruzione dell'`ArrayList` target, a partire dalla query, inserendo in esso tutti i match trovati. Come passo successivo veniva mostrato in output il numero di match totali trovati per poi, mediante un algoritmo che, come vedremo più avanti è computazionalmente molto inefficiente, venivano eliminati i match duplicati. A questo punto si mostrava in output il numero di match distinti ed, infine, si mostravano graficamente i match distinti (assieme agli identificatori dei nodi di ogni match).

La classe Java di Netmatch rappresentante il pannello principale del plugin è `netMatchPanel`. All'interno del pannello principale è possibile selezionare, nei modi elencati precedentemente, la query e il target su cui essa si vuole applicare. Premendo sul pulsante “Match” viene avviata la ricerca dei match. Alla pressione del pulsante “Match” viene creato un oggetto di tipo `netMatchTask` che si occuperà di ricercare i match. Il metodo della classe `netMatchTask` che si occupa delle ricerche dei match è `runMatch`. All'interno di tale metodo, a sua volta, viene creato un oggetto di tipo `netMatch` di cui viene invocato il metodo anch'esso chiamato `runMatch`. Tale metodo costruirà l'`ArrayList` contenente i match trovati. Il metodo `runMatch` della classe `netMatch` per prima cosa crea delle istanze di due `ArrayList` (`array` e `source`) che verranno passate come argomento al metodo `match` della classe `myMatch`. È proprio in questa classe che gli `ArrayList` `array` e `source` verranno popolati e verranno ricercati i vari match. Il

metodo `match` visita tutti i possibili match tra due grafi ed invoca, di volta in volta un metodo `vis` che costruisce, di volta in volta, un singolo match.

```
private boolean vis(int n, int ni1[], int ni2[], ArrayList array,
ArrayList source) {
    int tmp1[] = new int[n];
    int tmp2[] = new int[n];
    for(int i = 0; i < n; i++) {
        tmp1[i] = ni1[i];
        tmp2[i] = ni2[i];
    }
    array.add(tmp2);
    source.add(tmp1);
    return false;
}
```

(Nota: per ragioni di spazio sono omesse porzioni di codice non importanti ai fini della descrizione dell'algoritmo e la gestione delle eccezioni).

Il metodo `vis` viene invocato tante volte quanti sono i match che verranno trovati e si occupa di inserire il singolo match trovato nell'opportuno `ArrayList`. Come si può notare intuitivamente, il metodo è computazionalmente abbastanza semplice.

Terminata l'esecuzione del metodo `runMatch` della classe `netMatch` avremo terminato la costruzione dell'`ArrayList` contenente i match trovati. Invocando il metodo `getArrayDest` della classe `netMatchTask` all'interno della classe `NetMatchPanel` otterremo tale `ArrayList`. A questo punto, mostrato il numero totale dei match (cioè la dimensione dell'`ArrayList`) si invoca il metodo `eliminaDuplicati` di `netMatchPanel` per la rimozione dei match duplicati.

```
public void eliminaDuplicati(ArrayList lista) {
    int k=0;
    int j=1;
    for(int n=0; n<lista.size(); n++) {
        int[] array = (int[])lista.get(n);
        Arrays.sort(array);
    }
    while(k<(lista.size()-1)) {
        while(j<lista.size()) {
            int[] arrayKappa = (int[])lista.get(k);
            int[] arrayJay = (int[])lista.get(j);
            int count=0;
            for(int i=0; i<arrayKappa.length; i++) {
                if(arrayKappa[i] == arrayJay[i])
                    count++;
            }
            if(count == arrayKappa.length)
                j=j+1;
            else
                j=j+1;
        }
        k=k+1;
    }
}
```

```

        lista.remove(j);
    else
        j++;
    }
    k++;
    j=k+1;
}

```

Il criterio utilizzato è il seguente: si ordinano, in primo luogo gli array di interi rappresentanti i cammini. Risulta evidente, infatti, che i match equivalenti avranno gli stessi nodi, ma in ordine differente. Quindi è possibile eliminare i duplicati scorrendo l'ArrayList e, per il primo elemento eliminando eventuali duplicati a partire dal secondo elemento in poi, per il secondo elemento idem e così via. Anche in questo caso, intuitivamente, risulta evidente come tale metodo sia esageratamente oneroso. In effetti è proprio questo metodo a rendere assolutamente inefficiente l'algoritmo.

In primo luogo, dunque, la prima fase dell'attività di stage (successivamente a quella di studio delle classi principali di Netmatch di cui sopra si è data una descrizione) è consistita nell'analisi delle prestazioni del suddetto algoritmo e nella definizione di una possibile strategia differente che potesse abbattere radicalmente il costo dell'esecuzione di tale algoritmo. Come già anticipato prima il metodo `vis` viene invocato tante volte quanti sono i match (che corrispondono alla dimensione dell'ArrayList target). Supponiamo tale dimensione sia m . Quindi il metodo `vis` viene invocato m volte. Supponiamo inoltre che il numero medio di ogni cammino individuato sia pari ad n . Ovviamente n corrisponderà alla lunghezza media di ogni array di interi rappresentante il cammino. Il ciclo iniziale di `vis` (per la costruzione degli array) impiega tempo $O(n)$ più tempo $O(1)$ per l'aggiunta negli ArrayList. La complessità del metodo `match`, che invoca m volte `vis`, sarà dunque $O(mn)$ e, tutto sommato, abbastanza ragionevole. Il metodo `eliminaDuplicati` è composto da un primo ciclo `for` sulla dimensione dell'ArrayList (quindi m) in cui viene eseguito un ordinamento di ciascun array d'interi mediante il metodo `sort` della classe `Arrays` di `java.util` che, essendo un'implementazione del quicksort impiega tempo $O(n \log(n))$. Il metodo `get` invocato prima impiega tempo $O(1)$. Successivamente si hanno due cicli `for` annidati sulla dimensione dell'ArrayList in cui vengono eseguite due `get` (tempo $O(1)$) ed un ciclo `for` sulla dimensione di ogni array (tempo $O(n)$). In totale avremo quindi complessità $O(m^2 n)$ che, tenendo conto anche del primo ciclo, arriverà a $O(m^2 n \log(n))$. Il che è un risultato abbastanza irragionevole.

La strategia che si è inteso di adottare è stata quella di evitare l'invocazione del metodo `eliminaDuplicati` a costo, eventualmente, di rendere, entro margini abbastanza ragionevoli, leggermente più complesso il metodo `vis` per la determinazione dei match. L'idea di base è consistita

nell'utilizzare una struttura dati alternativa, le cui operazioni di ricerca e di inserimento fossero quanto più possibilmente efficienti in cui memorizzare temporaneamente i vari match trovati di volta in volta senza inserire i match duplicati. A partire da questa struttura, terminata la ricerca dei match, sarebbe stato necessario ricostruire l'ArrayList che, a questo punto, non avrebbe avuto più duplicati. La scelta della struttura dati da utilizzare è ricaduta sulla tabella hash. La struttura dati di questo tipo è implementata nel package `java.util` delle API di Java dalla classe `Hashtable`. È apparso fin da subito evidente che tale struttura dati costituisse il compromesso ideale in termini sia di occupazione di memoria che di tempi di esecuzione delle operazioni ([3]). Un'istanza della classe `Hashtable` creata con i parametri predefiniti (e quindi con un fattore di carico pari a 0,75) permette di ottenere il giusto compromesso tra dimensione della tabella e riduzione delle collisioni, il che permette di avere, in linea generale, tempi costanti sia per l'operazione di ricerca, implementata tramite il metodo `get`, che per l'operazione di inserimento, implementata tramite il metodo `put` ([2]). L'idea di base è stata, inizialmente, quella di costruire una tabella hash da passare ai vari metodi che conducevano a `vis` della classe `netMatch`. Al ritrovamento di un singolo match, si costruisce una stringa a partire dall'array rappresentante il cammino trovato inserendo come separatore per gli elementi un carattere speciale (ad esempio “#”), in modo da poterlo utilizzare come riferimento successivo per la ricostruzione dell'ArrayList a partire dalla `Hashtable`. Di volta in volta, quindi, il match trovato viene convertito in stringa ed inserito nella `Hashtable` solo se, effettivamente non ancora presente nella `Hashtable` (quindi se quel match non è già stato inserito ad una precedente invocazione di `vis`). Occorre quindi di volta in volta ordinare l'array all'interno del metodo `vis`.

```
//Ordinamento dell'array
Arrays.sort(tmp2);
//Creazione della stringa
String s = "";
for (int i = 0; i < n; i++) //Costruzione stringa
    s += Integer.toString(tmp3[i])+"#"; //Notare il carattere speciale
//Se la tabella non contiene la stringa la si inserisce
if (!table.containsKey(s)) {
    table.put (s,s);
}
```

A partire da una `Hashtable` così determinata, nella classe `netMatchPanel` non è più necessario invocare il metodo `eliminaDuplicati` in quanto la `Hashtable` non avrà nessun match duplicato. Occorre tuttavia implementare una nuova funzione `ricostuisciArray` che permette di ricostruire l'ArrayList a partire dalla `Hashtable`. Ciò risulta banale avendo usato un carattere speciale che può permettere di ricostruire l'array (con gli elementi già ordinati) a partire dalle singole stringhe. La complessità del metodo `vis` è aumentata: abbiamo un ciclo `for` per la costruzione della stringa ($O(n)$) ed un

ordinamento ($O(n \log(n))$). Complessivamente, tenuto conto che il metodo `match` viene invocato, come già detto, m volte, esso avrà complessità $O(mn \log(n))$: accettabile, tenuto conto che `eliminaDuplicati` non sarà invocata più. Banalmente il metodo `ricostruisciArray`, che verrà invocato una sola volta (come già accadeva per `eliminaDuplicati`), ricostruirà l'array a partire dalla stringa ed avrà complessità $O(mn)$ (per ogni elemento dell'`ArrayList` si effettua un ciclo per la costruzione di ogni singolo array).

Da notare che nell'implementazione si è preferito utilizzare stessa chiave e stesso valore nella tabella hash per fare in modo di poter recuperare agevolmente i valori tramite le chiavi.

In definitiva quello ottenuto sarebbe già un buon risultato, ma si può fare di meglio. Ad esempio evitare l'invocazione del metodo `ricostruisciArray` nella classe `netMatchPanel` inserendo direttamente i `match` non duplicati nell'`ArrayList` `target` nel metodo `vis`.

```
Arrays.sort(tmp2);
String s = "";
for (int i = 0; i < n; i++) //Costruzione stringa
    s += Integer.toString(tmp2[i]);
if (!table.containsKey(s)) {
    table.put (s,s);
    array.add(tmp2); //l'elemento non è presente, quindi lo si
inserisce direttamente
}
else {
    table.put (s,s);
}
source.add(tmp1);
```

Non è più necessario utilizzare il carattere separatore speciale perché adesso non è più necessario ricostruire l'`ArrayList` a partire dall'`Hashtable` in quando gli elementi vengono inseriti di volta in volta, ossia l'array viene inserito nell'`Hashtable` ogni qualvolta è verificata la prima condizione. La complessità di `vis` non cambia, ma in compenso non è necessario più il metodo `ricostruisciArray` prima implementato.

Poiché l'`ArrayList` che viene restituito dal metodo `getArrayDest` della classe `netMatch` ed invocato in `netMatchPanel` restituisce, con questa nuova implementazione, l'`ArrayList` definitivo e senza duplicati non possiamo tenere traccia, come era possibile prima, del numero di `match` totali, che come già detto, veniva originariamente mostrato in output. Implementando un apposito contatore nella funzione `vis` è possibile tenere traccia di tutte le sue chiamate e quindi di tutti i `match` che vengono trovati. È possibile mostrare in output il numero totale di `match` chiamando una funzione `getTotalMatches`

opportunamente implementata in `netMatchTasks`.

Il primo obiettivo è, quindi, stato raggiunto. Tuttavia in Netmatch viene data la possibilità all'utente non solo di costruire delle query precise da passare in input, ma, utilizzando l'apposito pannello detto Query Editor ed implementato mediante la classe `QtoolFrame`, è possibile anche costruire delle query contenenti dei cammini approssimati ([1]). È possibile poter dunque ricercare i match contenenti cammini di qualsiasi lunghezza. L'utente ha la possibilità di impostare come meglio predilige la lunghezza di ogni singolo cammino. Una volta trovati i match (tramite l'invocazione del metodo `match`) all'interno della classe `netMatch` si verifica se sono stati trovati effettivamente dei match e si controlla se sono presenti nella query cammini approssimati. Se sono presenti occorre trovare tutti i cammini richiesti, quindi occorre ricostruire l'`ArrayList` target tenendo conto di essi. La ricerca dei cammini viene effettuata mediante un apposito metodo `bfs` che effettua una visita sul grafo. Tuttavia la nuova implementazione dell'algoritmo forniva, a tale passo, l'`ArrayList` già privo di tutti i duplicati e con ciascun elemento già ordinato. Di conseguenza l'eventuale ricerca dei cammini falliva inevitabilmente dando in output risultati incoerenti. È stato dunque necessario modificare ulteriormente il metodo `vis`. In particolar modo occorre mantenere anche traccia del ritrovamento di tutti match nell'ordine esatto in cui essi venivano trovati e compresi di duplicati. Quindi è stata utilizzata una nuova struttura di tipo `ArrayList` che mantenesse traccia di ciò. In questo caso è stato necessario implementare in `vis` un nuovo inserimento mediante metodo `add`. Ma essendo tale operazione eseguibile in tempo $O(1)$ ciò non ha comportato un costo in termini di prestazioni (se non in termini di memoria occupata). Successivamente all'invocazione di `match` nella classe `netMatch` si verifica (mediante un apposito controllo) se sono stati trovati match (si controlla un apposito contatore) e se la query contiene cammini approssimati (mediante un controllo sulla variabile booleana `isQueryApprossimate` implementata appositamente). Se la condizione è verificata si costruisce un nuovo `ArrayList` di destinazione ottenuto tenendo conto dei cammini approssimati. Al termine dell'esecuzione della ricerca mediante chiamata al metodo `setArrayDest` si aggiorna il nuovo `ArrayList`. Queste operazioni venivano effettuate sull'`ArrayList` di destinazione ottenuto mediante chiamata al metodo `match`. Come detto tale `ArrayList` si prevedeva avesse i duplicati (che poi sarebbero stati eliminati con il metodo già visto precedentemente). Stavolta si costruisce il nuovo `ArrayList` a partire da quello di appoggio costruito appositamente e costruito nelle chiamate al metodo `vis` perché l'`ArrayList` di destinazione ottenuto non può essere più utilizzato a tale scopo. Anche in questo caso occorre, prima di inserire ogni singolo elemento nell'`ArrayList` costruire la corrispondente stringa (in modo analogo a quanto visto precedentemente) ed inserire l'elemento solo se necessario. Sono qui mostrate le parti iniziale e finale della porzione di codice specifica che controlla l'esistenza di cammini approssimati e poi effettua l'inserimento vero e proprio.

```

if(val.intValue() > 0 && isApproximate) {
    ArrayList l1 = getArraySource();
    ArrayList l2 = getArrayDest2();
    Hashtable t = new Hashtable();
    ArrayList l3 = new ArrayList();
    ArrayList l4 = new ArrayList();

/* ALGORITMO PER IL CALCOLO DEI CAMMINI (codice omissso) */

if(count == qPaths.size()) {
    l3.add(tmp1);
    //l4.add(tmp2); //Prima veniva effettuato l'inserimento
diretto
    total++;
    Arrays.sort(tmp2);
    String s = "";
    for (int i = 0; i < tmp2.length; i++)
        s += Integer.toString(tmp2[i]);
    if (!t.containsKey(s)) { //Eventuale inserimento in l4
        t.put (s,s);
        l4.add(tmp2);
    }
    else {
        t.put (s,s);
    }
}

```

Come si può notare `l2` conterrà l'ArrayList di appoggio e non quello di destinazione, deve essere creata una nuova tabella hash nella quale verranno inserite le stringhe costruite. Alla fine possono essere aggiornati gli ArrayList.

```

setArrayDest(l4);
setArraySource(l3);

```

In buona sostanza, con questa nuova implementazione, si è ridotto di un ordine di grandezza la complessità dell'algoritmo. Considerato che, tendenzialmente, le operazioni di ricerca ed inserimento nella tabella hash sono costanti, è possibile fare svariate considerazioni: il metodo `vis` viene invocato tante volte quanti sono i match totali (m volte, in base a come supposto precedentemente); il metodo `vis` definitivo è composto da un primo ciclo `for` per la costruzione degli array ($O(n)$), la creazione di una copia dell'array da mantenere non ordinato ($O(n)$), un'operazione di ordinamento ($O(n \log(n))$), la creazione della stringa ($O(n)$); a questo punto si controlla se la stringa è presente nella tabella hash ($O(1)$); se è presente viene effettuato un inserimento ($O(1)$), se non è presente viene aggiornato solo il contatore, quindi occorre effettuare una ricerca nella tabella (pur sempre tempo $O(1)$). In definitiva la complessità

di vis è $O(n\log(n))$. Essendo eseguito in totale m volte avremo $O(mn\log(n))$ a fronte di $O(m^2n\log(n))$. Dal momento in cui, in generale, Cytoscape viene utilizzato per l'analisi di reti biologiche molto complesse e, generalmente, aventi moltissime occorrenze duplicate, è, come sarà mostrato più avanti, un ottimo risultato.

Supponiamo, adesso, che ad un determinato passo dell'algoritmo vengano inseriti i seguenti identificatori di nodi del grafo nella tabella hash (i valori sono, in questo caso, di pura fantasia): {1,2,3}, {4,6,7}, {5,9}.

La tabella hash generata avrà la struttura mostrata nella tabella che segue.

Key	Value
123	123
467	467
59	59

Tabella 1: Il contenuto della tabella hash durante l'esecuzione di Netmatch

Le chiavi ed i corrispondenti valori sono stringhe costruite a partire dagli array d'interi. Dal momento in cui questa implementazione era stata studiata per il caso iniziale in cui volevamo ricostruire in un secondo momento l'ArrayList a partire dalla Hashtable (originariamente avevamo inserito un carattere speciale come separatore per poter ricostruire ogni array), a questo punto questo espediente risulta fondamentalmente superfluo. È possibile però sfruttare le caratteristiche di questa struttura per mantenere memorizzati, per ogni singolo match il numero esatto di occorrenze che sono state trovate, quindi mantenere come chiave della tabella hash la stringa (necessario per effettuare il controllo pre-inserimento) e come valore, invece, incrementare di volta in volta un apposito contatore (sfruttando la classe Integer). Quindi in corrispondenza dei controlli pre-inserimento è stata effettuata la seguente modifica:

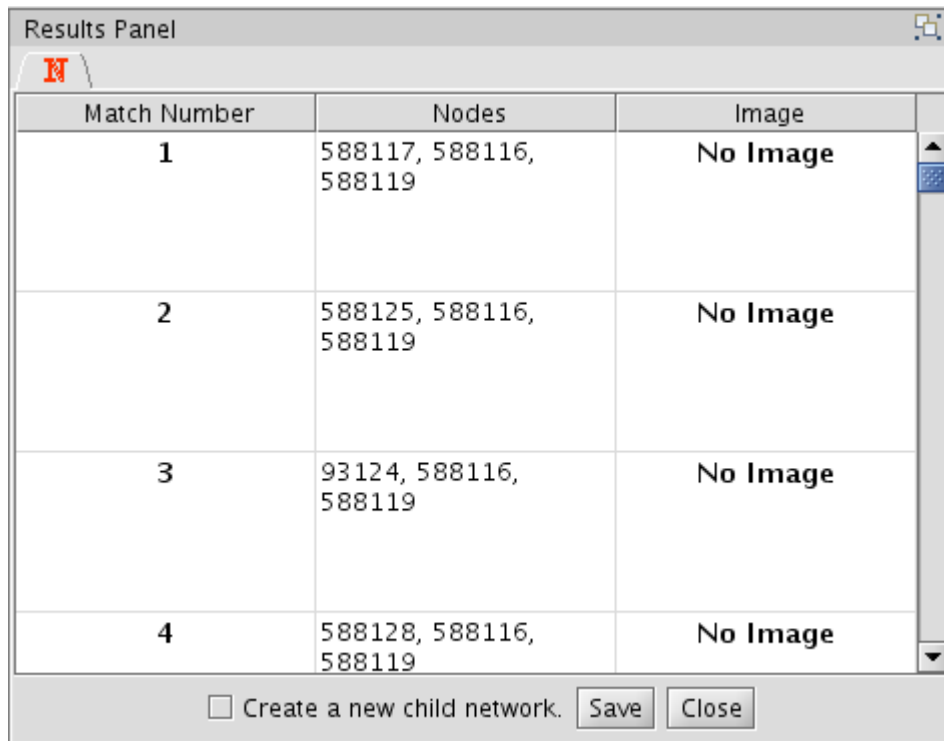
```
if (!table.containsKey(s)) {
    table.put (s,new Integer(1));
    array.add(tmp3);
}
else {
    table.put (s,new Integer(((Integer)table.get(s)).intValue()+1));
}
```

Sopra è mostrata l'implementazione nel metodo vis, analogo procedimento vale nella porzione di codice per il calcolo dei cammini approssimati. Il perché di questa scelta sta nel fatto che, rispetto alla versione 2.1 di Netmatch, si vogliono mostrare in output, oltre che all'elenco di match con gli identificatori di nodo ed (eventualmente) la rappresentazione grafica, anche il numero di occorrenze ripetute per ogni singolo match.

La rappresentazione in output dei risultati avviene mediante la classe `netMatchResultTableModel`. La complicazione sta, tuttavia, nel fatto che in questa classe poco prima della rappresentazione grafica dei risultati i grafi vengono convertiti nel formato riconosciuto da Cytoscape. Del resto, in output, si vogliono mostrare gli identificatori dei nodi così come sono riconosciuti da Cytoscape. Risulta quindi necessario associare ai nuovi identificatori il numero di occorrenze corrispondente ma associato ai corrispondenti identificatori nel formato usato da Netmatch.

```
int tmp[] = (int[])complexes.get(i);
String s = "";
for (int j=0; j<tmp.length; j++)
    s += tmp[j];
String dupl =
Integer.toString(((Integer)table.get(s)).intValue());
data[i][0] = (new Integer(i + 1)).toString();
data[i][1] = dupl;
data[i][2] = getNodeNameList(gpComplex);
data[i][3] = "No Image";
```

`complexes` contiene i nuovi identificatori nel formato di Cytoscape, mentre `s` è la stringa corrispondente, ma riferita ai vecchi identificatori; `dupl` è la stringa contenente il numero di match e `data[i][1]` è una nuova colonna aggiunta nella rappresentazione tabulare dell'output.



Match Number	Nodes	Image
1	588117, 588116, 588119	No Image
2	588125, 588116, 588119	No Image
3	93124, 588116, 588119	No Image
4	588128, 588116, 588119	No Image

☐ Create a new child network. Save Close

Illustrazione 3: Tabella con output senza visualizzazione del numero di occorrenze

Results Panel			
Match Number	Occurrences	Nodes	Image
1	2	588117, 588116, 588119	No Image
2	6	588125, 588116, 588119	No Image
3	6	93124, 588116, 588119	No Image
4	2	588128, 588116, 588119	No Image

☐ Create a new child network. Save Close

Illustrazione 4: Tabella con output con visualizzazione del numero di occorrenze

4. Testing

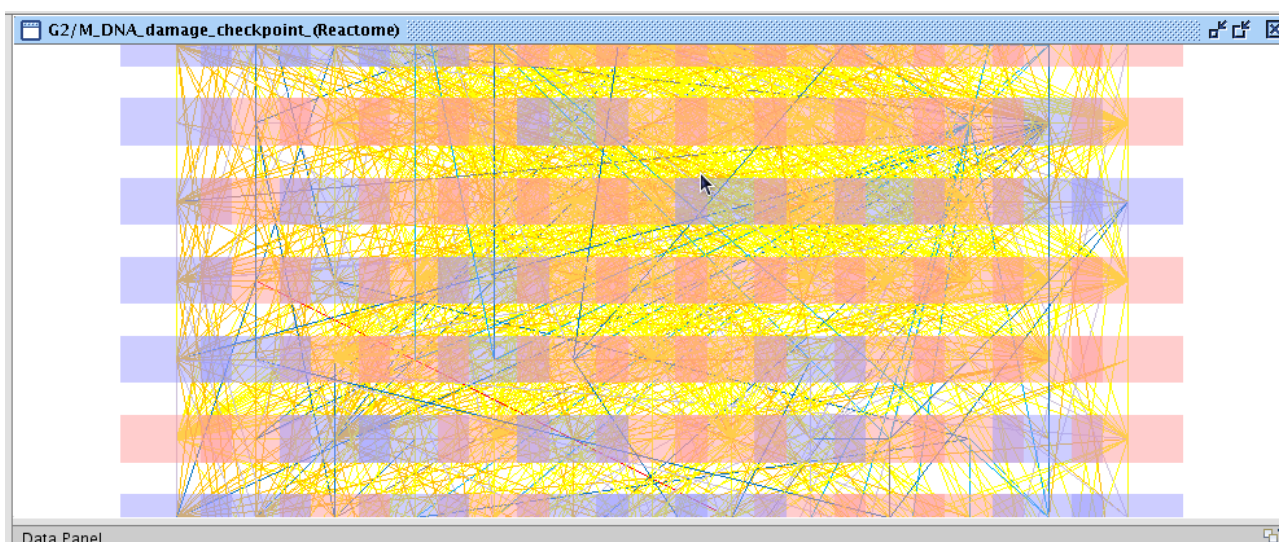
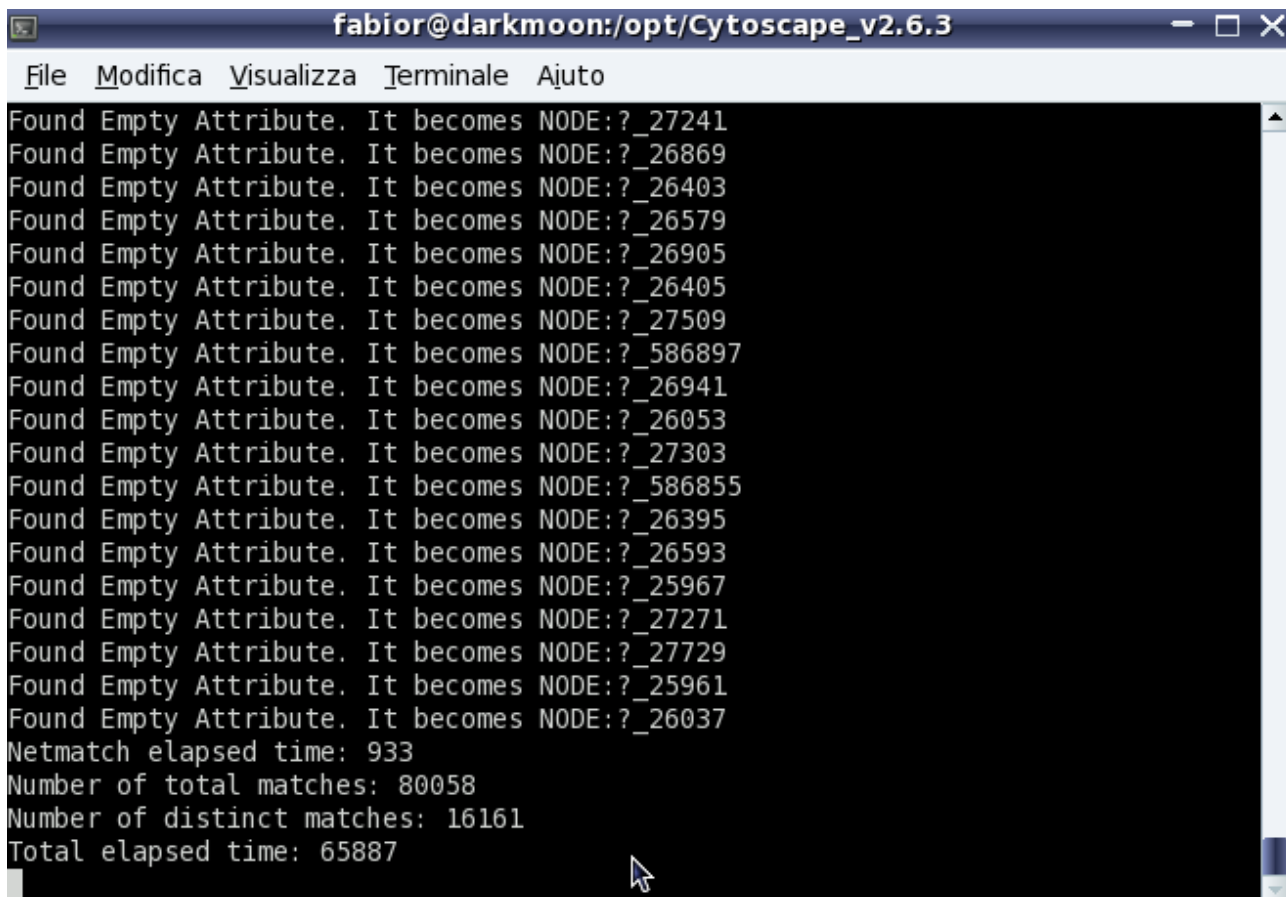


Illustrazione 5: Una rete usata come esempio per il testing

La prima fase dell'attività di testing è consistita nella verifica della correttezza dell'algoritmo implementato. Utilizzando grafi personalizzati in un primo momento abbastanza semplici, poi il grafo d'esempio galFiltered reso disponibile all'installazione di Cytoscape e, come passo finale, complesse reti scaricate da

appositi database resi disponibili in internet, si è verificato innanzitutto che sia la versione 2.1 del plugin che la nuova versione aggiornata ottenessero gli stessi risultati, sia per quanto riguarda il numero di match totali, sia per quelli distinti e sia per l'esatto elenco di match ottenuti in output. Si è poi verificato che la somma delle occorrenze duplicate di ogni singolo match corrispondesse al numero totale di match individuati.

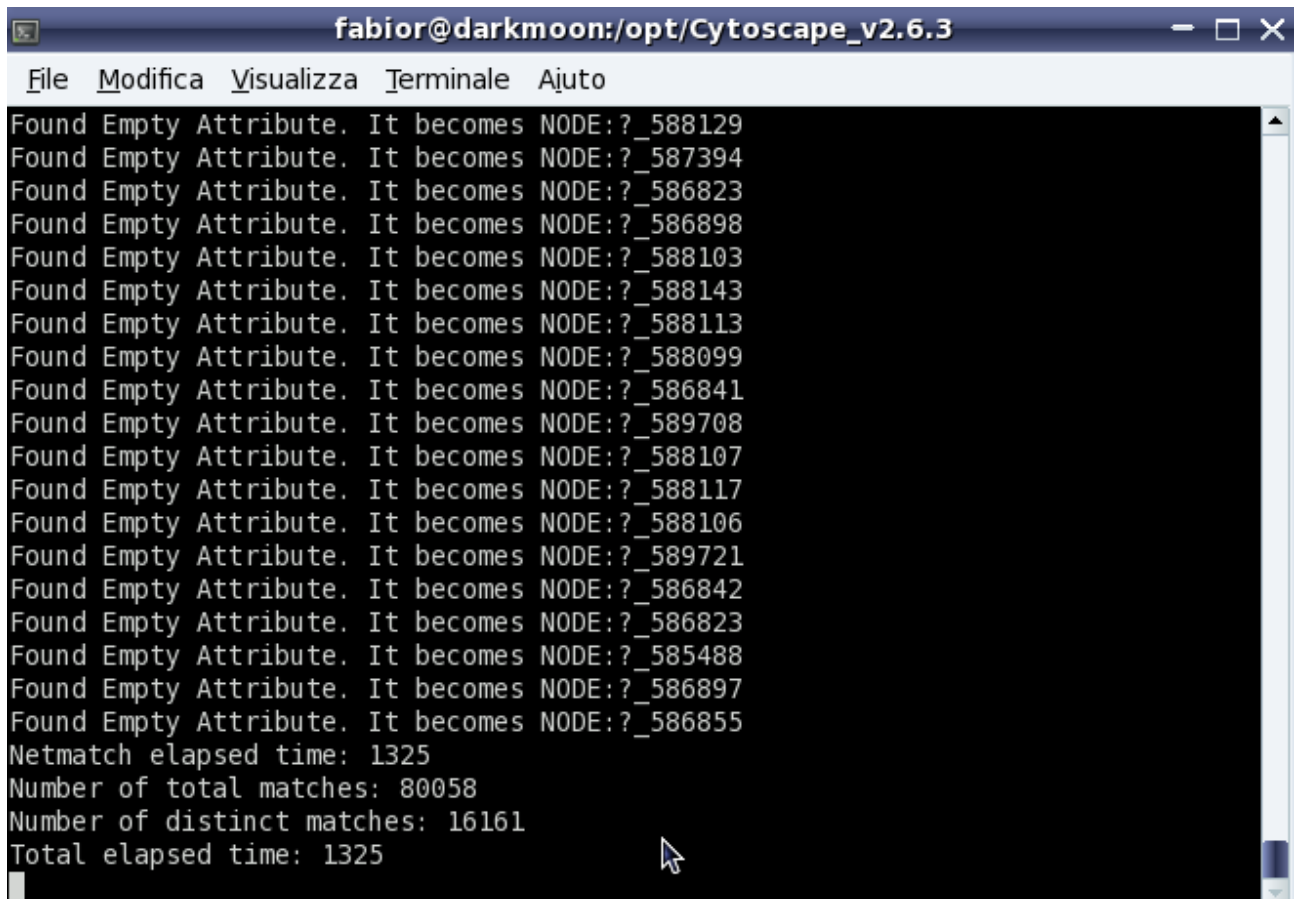


```
fabior@darkmoon:/opt/Cytoscape_v2.6.3
File Modifica Visualizza Terminale Ajuto
Found Empty Attribute. It becomes NODE:?_27241
Found Empty Attribute. It becomes NODE:?_26869
Found Empty Attribute. It becomes NODE:?_26403
Found Empty Attribute. It becomes NODE:?_26579
Found Empty Attribute. It becomes NODE:?_26905
Found Empty Attribute. It becomes NODE:?_26405
Found Empty Attribute. It becomes NODE:?_27509
Found Empty Attribute. It becomes NODE:?_586897
Found Empty Attribute. It becomes NODE:?_26941
Found Empty Attribute. It becomes NODE:?_26053
Found Empty Attribute. It becomes NODE:?_27303
Found Empty Attribute. It becomes NODE:?_586855
Found Empty Attribute. It becomes NODE:?_26395
Found Empty Attribute. It becomes NODE:?_26593
Found Empty Attribute. It becomes NODE:?_25967
Found Empty Attribute. It becomes NODE:?_27271
Found Empty Attribute. It becomes NODE:?_27729
Found Empty Attribute. It becomes NODE:?_25961
Found Empty Attribute. It becomes NODE:?_26037
Netmatch elapsed time: 933
Number of total matches: 80058
Number of distinct matches: 16161
Total elapsed time: 65887
```

Illustrazione 6: Tempi di esecuzione della query dell'illustrazione 2 su Netmatch 2.1

La fase più importante di testing è stata quella relativa all'efficienza dell'algoritmo. Nel capitolo precedente è stata mostrata la complessità computazionale della versione 2.1 di Netmatch e della nuova versione aggiornata (che al termine della fase di testing sarà la versione 2.2). È particolarmente interessante valutare in termini di millisecondi di esecuzione il miglioramento delle prestazioni della nuova implementazione. Risulta fin da subito evidente che per query che restituiscono un numero di match totali e di match distinti dello stesso ordine di grandezza, l'algoritmo risulta essere di poco meno efficiente. Del resto ciò dipende dall'ordinamento che viene eseguito in `vis`. Tuttavia se, il numero di match totali e ragionevolmente più grande del numero di match distinti, i risultati sono sbalorditivi. Provando sulla network G2/M_DNA_damage_checkpoint_(Reactome) importata da apposito Database e ricercando una semplice query costituita da tre nodi e tre cammini si può verificare che si ottengono 80058 match totali e 16161 match distinti. Con la versione 2.1 di Netmatch, il tempo di esecuzione (in millisecondi) per la ricerca dei

match totali è stato pari a 933; il tempo totale (aggiungendo il tempo per l'eliminazione dei duplicati) è stato addirittura pari a 65887. Con la versione 2.2, il tempo di esecuzione per la ricerca dei match totali è stato 1325 (incremento molto accettabile considerato il numero abbastanza alto di match ricercati) ed il tempo totale di esecuzione pari sempre a 1325: del resto, l'eliminazione dei duplicati è già avvenuta nei 1325 millisecondi iniziali e non successivamente. Quindi, in definitiva, 1325 millisecondi a fronte di 65887: poco più di un secondo rispetto a ben oltre un minuto.



```
fabior@darkmoon:/opt/Cytoscape_v2.6.3
File Modifica Visualizza Terminale Ajuto
Found Empty Attribute. It becomes NODE:?_588129
Found Empty Attribute. It becomes NODE:?_587394
Found Empty Attribute. It becomes NODE:?_586823
Found Empty Attribute. It becomes NODE:?_586898
Found Empty Attribute. It becomes NODE:?_588103
Found Empty Attribute. It becomes NODE:?_588143
Found Empty Attribute. It becomes NODE:?_588113
Found Empty Attribute. It becomes NODE:?_588099
Found Empty Attribute. It becomes NODE:?_586841
Found Empty Attribute. It becomes NODE:?_589708
Found Empty Attribute. It becomes NODE:?_588107
Found Empty Attribute. It becomes NODE:?_588117
Found Empty Attribute. It becomes NODE:?_588106
Found Empty Attribute. It becomes NODE:?_589721
Found Empty Attribute. It becomes NODE:?_586842
Found Empty Attribute. It becomes NODE:?_586823
Found Empty Attribute. It becomes NODE:?_585488
Found Empty Attribute. It becomes NODE:?_586897
Found Empty Attribute. It becomes NODE:?_586855
Netmatch elapsed time: 1325
Number of total matches: 80058
Number of distinct matches: 16161
Total elapsed time: 1325
```

Illustrazione 7: Tempi di esecuzione della query dell'illustrazione 2 su Netmatch 2.2

5. Bugs

La parte finale dell'attività di stage è consistita, invece, nell'individuazione di piccoli bug, già presenti nel plugin ancor prima che venissero effettuate le modifiche descritte nei capitoli precedenti. In particolar modo è stato individuato un bug nel QueryEditor: ogni qual volta veniva disegnata una query si mantenevano le informazioni sul numero di nodi, di cammini e di cicli generati nella query. Se la query rappresentata veniva passata a Netmatch e successivamente il Query Editor veniva riavviato (ad esempio per disegnare un'ulteriore query) i precedenti valori non venivano azzerati, causando incoerenze, non rilevanti ai fini della ricerca dei match ma pur sempre deprecabili.

6. Note conclusive

L'IDE usato per la modifica del codice Java è stato NetBeans 6.7.1 con JDK 1.6.0_16 su Mandriva Linux 2010.0 Free (Linux 2.6.31.6-desktop-1mnb). La versione di Cytoscape utilizzata è stata la 2.6.3. La presente relazione è stata scritta ed impaginata con OpenOffice 3.1.1.

Bibliografia

- [1] A. Ferro, R. Giugno, G. Pigola, A. Pulvirenti, D. Skiprin, G. D. Bader, D. Shasha. *NetMatch: a Cytoscape plugin for searching biological networks*. *Bioinformatics* (2007) pagg. 910-912
- [2] <http://java.sun.com/javase/6/docs/api/index.html>. *Java Platform API Specification* (2002)
- [3] T. Cormen, C. Leiserson, R. Rivest, C. Stein. *Introduction to algorithms* (2002) pagg. 229-231