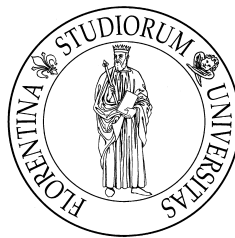


UNIVERSITÀ DEGLI STUDI DI FIRENZE
Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica



Tesi di Laurea

ANALISI DI RETI METABOLICHE BASATA SU
PROPRIETÀ DI CONNESSIONE

MASSIMO NOCENTINI

Relatore: *Pierluigi Crescenzi*

Anno Accademico 2010-2011

INDICE

1	Introduzione	3
1.1	Enzimi, cammini e reti metaboliche	3
1.2	Il linguaggio SBML	4
1.2.1	Oggetti e proprietà fondamentali	4
1.2.2	Costruire un grafo da un modello SBML	5
1.3	Riferimenti a lavori esistenti	7
1.4	Nostri obiettivi	7
1.5	Sinopsi	8
1.6	Strumenti utilizzati	9
2	Algoritmi	11
2.1	Alcune definizioni	11
2.2	Visitare un grafo	12
2.2.1	Il concetto di visita	12
2.2.2	Visita in profondità	12
2.2.3	Pseudo codice e complessità	13
2.2.4	Punti di estensione	14
2.2.5	Costruzione dell'albero DFS	14
2.3	Ricerca delle componenti fortemente connesse	15
2.3.1	Kosaraju: componenti pozzo e inversione del grafo	15
2.3.2	Tarjan: una pila e la funzione lowlink	16
2.3.3	Algoritmo di Tarjan mediante due pile	19
3	Studio	21
3.1	Funzionalità	21
3.1.1	Costruzione del grafo a partire da una codifica SBML	21
3.1.2	Rappresentazione grafica del grafo con nodi bianchi e neri	21
3.1.3	Applicare la visita in profondità	23
3.1.4	Applicare l'algoritmo di Tarjan	23
3.1.5	Rappresentazione tabulare	24
3.1.6	Ridurre la complessità collassando i nodi sorgente	27
3.1.7	Indagare proprietà di una batteria di modelli	28
3.1.8	Analizzare un grafo non associato ad un modello SBML	31
3.2	L'architettura <i>pipes and filters</i>	31
3.2.1	Cenni teorici	31
3.2.2	Implementazione in questo progetto	33
4	Esperimenti e conclusioni	35
4.1	Risultati	35
4.1.1	Modelli biologici studiati	35
4.1.2	Come riprodurre le nostre esperienze	35

4.1.3	Particolarità di alcuni modelli	38
4.1.4	Osservazioni su quanto ottenuto	38
4.2	Sviluppi futuri	41
5	Appendice	45
5.1	Metodologia di sviluppo adottata	45
5.1.1	Learning tests	45
5.2	Descrizione dei pacchetti del progetto	46
5.2.1	Pacchetto dotInterface	46
5.2.2	Pacchetto JSBMLInterface	47
5.2.3	Pacchetto Model	48
5.2.4	Pacchetto Piping	51
5.2.5	Pacchetto Tarjan	53

INTRODUZIONE

Questo lavoro ha come obiettivo l'analisi di reti metaboliche: cercheremo di astrarre i processi fisici che avvengono all'interno delle cellule di organismi, utilizzando la struttura astratta grafo in modo da poter ragionare su quest'ultimo utilizzando algoritmi ben noti della teoria dei grafi. In particolare indagheremo la relazione di connessione forte, ricercando le componenti fortemente connesse.

In questo capitolo si definisce cosa sono le reti metaboliche, il motivo per cui vengono studiate, come formulare un modello astratto che descriva una rete, quali sono i risultati già noti che condividono alcuni aspetti di questo lavoro e quali sono gli obiettivi che vogliamo raggiungere.

1.1 ENZIMI, CAMMINI E RETI METABOLICHE

Prima di dare la definizione di *rete metabolica* introduciamo i concetti di cammino metabolico, enzima e da cosa è caratterizzata una reazione chimica.

Un *cammino metabolico* è una sequenza di reazioni chimiche che si verificano all'interno di una cellula. Dal punto di vista matematico, possiamo vedere un cammino metabolico come una sequenza di funzioni, dette reazioni, che, avendo in input un insieme di molecole, esegue delle trasformazioni su queste e produce come output il risultato delle trasformazioni svolte, sotto forma di insieme di molecole. Questo output può essere utilizzato come input per la funzione successiva, concatenando quanto si voglia le trasformazioni.

Ogni reazione chimica è regolata da alcuni *enzimi*. Un *enzima* è una proteina che gestisce la frequenza e la velocità di una reazione chimica. Le molecole a cui si applica la reazione vengono identificate con il termine *substrato* (o *reagenti*), mentre le molecole output della reazione vengono identificate con il termine *prodotti*. Durante l'esecuzione di una reazione chimica, ogni enzima agisce da *catalizzatore*, ovvero non viene consumato nella reazione e, quindi, può partecipare in più di una reazione.

L'insieme di enzimi "guida" e determina l'insieme di cammini metabolici che possono occorrere nella cellula, in quanto una reazione chimica su un substrato può avvenire se e solo se lo strato attivo del substrato è complementare allo strato attivo dell'enzima.

Adesso possiamo definire una *rete metabolica* come collezione di cammini meta-

bolici. Le reti metaboliche sono ampiamente studiate in quanto caratterizzano le proprietà fisiologiche e biochimiche delle cellule. Analizzare queste reti permette la ricostruzione delle reazioni biochimiche che avvengono all'interno di molti organismi, sia questi batteri che esseri umani.

Nelle prossime sezioni studieremo un linguaggio che permetta di codificare le reti metaboliche, descrivendo in particolare i concetti necessari al nostro lavoro.

1.2 IL LINGUAGGIO SBML

SBML (**S**ystems **B**iology **M**arkup **L**anguage) è un linguaggio che permette di rappresentare informazioni classificandole in modo gerarchico, basato sul linguaggio XML.

SBML è orientato alla descrizione di sistemi in cui entità biologiche sono oggetto di manipolazioni eseguite da processi nel corso del tempo e, pertanto, facilita la codifica di modelli computazionali di processi biologici, come, ad esempio reti metaboliche, segnalazioni cellulari e molti altri.

Nel seguito, dopo aver formalizzato un sistema con questo linguaggio, faremo riferimento a tale formalizzazione con il termine “modello SBML” del sistema in questione.

1.2.1 Oggetti e proprietà fondamentali

Nella precedente sezione abbiamo introdotto, per inquadrare il problema, alcuni concetti che non sono influenti sul nostro studio: in questa sezione trattiamo solo quelli inerenti al lavoro che abbiamo sviluppato, nonostante molti dei concetti non usati siano modellabili con SBML.

L'unità atomica definibile con SBML è il *metabolito*, che rappresenta il concetto di molecola. Un oggetto di questo tipo ha molti attributi ma, ai nostri fini, tre sono quelli necessari:

IDENTIFICATORE rappresentato da un'etichetta, univoca in tutto il modello, permette di distinguerlo dagli altri metaboliti;

NOME rappresentato da un'etichetta, possibilmente non univoca, permette di associargli delle informazioni di più alto livello rispetto all'*identificatore*;

COMPARTIMENTO rappresentato da un'etichetta, univoca in tutto il modello, permette di associargli il compartimento della cellula dove risiede (in tutti gli esempi che abbiamo avuto modo di testare il compartimento è sempre il *citoplasma*).

Un altro concetto fondamentale è quello di *reazione chimica*, codificato in SBML con:

REAGENTI insieme di *metaboliti*, modella il substrato della reazione;

PRODOTTI insieme di *metaboliti*, modella i prodotti della reazione;

REVERSIBILE valore booleano, specifica se la reazione è reversibile oppure no.

Questo è quello che ci serve per iniziare ad analizzare il modello SBML di un organismo: ricercheremo l'insieme di reazioni descritte e, per ogni reazione, analizzeremo l'insieme dei *reagenti* e dei *prodotti* per costruire un grafo oggetto dei nostri studi. Nella prossima sezione descriveremo le regole che abbiamo utilizzato per costruirlo.

1.2.2 Costruire un grafo da un modello SBML

Non è possibile utilizzare direttamente i concetti espressi con il linguaggio SBML come input per i nostri algoritmi, dobbiamo prima trasportarli in una struttura dati grafo. Questo passaggio ci permette di ridurre la complessità e la mole delle informazioni, riducendo il problema originale ad un problema a cui è possibile applicare algoritmi appartenenti alla teoria dei grafi.

In particolare, il grafo che costruiremo è un grafo orientato, la cui caratteristica è quella di partizionare l'insieme dei vertici in due partizioni. Identificheremo i vertici a seconda della partizione di appartenenza come vertici *neri* o *bianchi* (vedi [5]).

Per costruire il grafo usiamo questo insieme di regole:

- due specie sono uguali se hanno uguale identificatore ed uguale compartimento. Questa regola è necessaria per evitare un'esplosione del numero di vertici in quanto, date due reazioni r, r' e due metaboliti a, a' tali che $a \in \text{reagenti}(r) \wedge a' \in \text{reagenti}(r')$, con $a = a' \wedge r \neq r'$, se si costruissero due nodi distinti rispettivamente per a e a' il grafo non avrebbe informazioni significative, perché degenererebbe ad un insieme di sotto grafi isolati, ognuno rappresentante una reazione. Da questo segue che un metabolito non è identificato dalle reazioni in cui appare;

- data una reazione non reversibile r tale che:

$$\text{reagenti}(r) = \{r_1, \dots, r_n\}$$

$$\text{prodotti}(r) = \{p_1, \dots, p_m\}$$

allora costruiremo il grafo che codifica la relazione $\text{reagenti}(r) \times \text{prodotti}(r)$.

Ad esempio, con $\text{reagenti}(r) = \{a, b, c, d\}$ e $\text{prodotti}(r) = \{a, e, f\}$ otterremmo il grafo riportato in Figura 1;

- data una reazione reversibile r tale che:

$$\text{reagenti}(r) = \{r_1, \dots, r_n\}$$

$$\text{prodotti}(r) = \{p_1, \dots, p_m\}$$

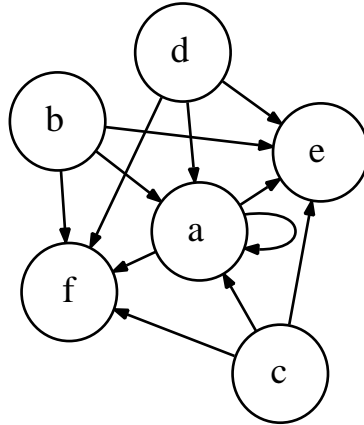


Figura 1: Grafo ottenuto da una reazione non reversibile

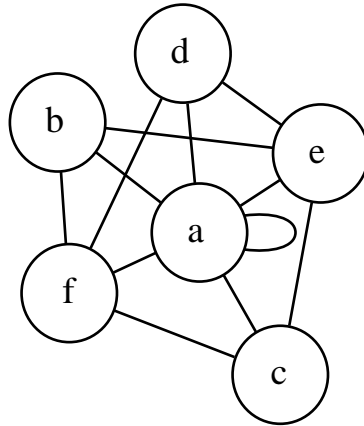


Figura 2: Grafo ottenuto da una reazione reversibile

allora costruiremo il grafo che codifica la relazione $(\text{reagenti}(r) \times \text{prodotti}(r)) \cup (\text{prodotti}(r) \times \text{reagenti}(r))$. Ad esempio, con $\text{reagenti}(r) = \{a, b, c, d\}$ e $\text{prodotti}(r) = \{a, e, f\}$ otteniamo il grafo riportato in Figura 2.

Le regole descritte sopra pongono dei limiti sulle informazioni che si codificano nel grafo finale. Per esempio tutti gli attributi delle reazioni descritti in SBML vengono ignorati. Questo non rende insufficiente la nostra rappresentazione in quanto miriamo alla relazione di connessione estratta dalle reazioni e non a loro particolari proprietà.

Dobbiamo essere coscienti che applicando le precedenti regole facciamo un passo di astrazione che comporta la perdita di precisione: nel risultante grafo avremo codificato per ogni metabolito l'insieme di metaboliti in cui questo può essere trasformato, a prescindere dalle particolari reazioni che definiscono le trasformazioni.

1.3 RIFERIMENTI A LAVORI ESISTENTI

Il lavoro esposto in [6] ha come obiettivo lo studio di reti metaboliche per identificare quelli che vengono chiamati nell'articolo originale *seed sets*: questi rappresentano degli insiemi di molecole che sono delle "interfacce" tra la rete metabolica e il contesto che la ospita. Una volta identificati questi insiemi, si procede ad analizzarli e trarre conclusioni di natura biologica.

Abbiamo citato questo lavoro in quanto condivide parte del processo di astrazione della rete e l'applicazione di algoritmi con il presente documento.

Il nostro lavoro ha come obiettivo di essere d'ausilio al lavoro esposto in [5], che studia una variante del problema di enumerare tutti i sotto grafi aciclici di un grafo orientato, aggiungendo il vincolo che solo un determinato sottoinsieme \mathbb{B} di vertici possono avere il ruolo di sorgente o di pozzo nei *DAG* enumerati. Ogni *DAG* che soddisfa tale condizione è chiamato *storia* e si vuole "raccontare" tutte le possibili storie dato un grafo orientato e l'insieme \mathbb{B} . Ricordiamo che un *DAG* è un grafo orientato che non contiene cicli.

1.4 NOSTRI OBIETTIVI

Il concetto catturato dall'insieme \mathbb{B} , introdotto nella precedente sezione, è quello di definire il ruolo che alcuni vertici "interpreteranno" nella *storia* (se $v \in \mathbb{B}$ allora v avrà il ruolo di *sorgente* o *pozzo*, altrimenti il ruolo di *intermedio*).

Attualmente l'insieme \mathbb{B} viene costruito in base a osservazioni e studi *empirici*, ed è costituito dall'insieme di metaboliti che hanno particolari proprietà in determinate condizioni. Ci chiediamo se è possibile identificare l'insieme \mathbb{B} in modo automatico per analizzare la rete metabolica utilizzando gli algoritmi esposti in [5].

Per rispondere a tale domanda abbiamo deciso di analizzare la connettività del grafo in input, ricercando le sue componenti fortemente connesse. Questa idea si basa sulle seguenti motivazioni:

- è difficile cercare di assegnare il ruolo ad ogni vertice studiando l'istanza completa del grafo in input, in quanto astrarre una rete metabolica comporta grafi con molti vertici e numerosi archi ed, inoltre, possono esistere (con molta frequenza) cicli che portano solo ridondanza e complessità, che quindi possiamo escludere in quanto non aggiungono informazioni.

Identificando invece la decomposizione in componenti fortemente connesse, è possibile astrarre dai cicli ed identificare classi di metaboliti equivalenti che possono essere visti come uno solo;

- come vedremo nella Sezione 2.1, ogni componente fortemente connessa è una classe di equivalenza della relazione di connessione forte.

Ciò significa che due metaboliti appartenenti alla stessa componente fortemente connessa possono essere ottenuti l'uno dall'altro, attraverso una catena di reazioni nella rete metabolica che stiamo analizzando. Possiamo quindi associare a questi metaboliti lo stesso ruolo che viene associato alla componente che li contiene;

- il “meta grafo” che ha come vertici le componenti fortemente connesse e come archi gli archi che collegano metaboliti contenuti in componenti diverse (evitando duplicazioni), permette di assegnare alle componenti un ruolo. Se una componente è *sorgente* o *pozzo* nel meta grafo allora possiamo inserire i vertici che la compongono nell'insieme \mathbb{B} .

Utilizzando quanto detto sopra procederemo ad applicare al grafo in input l'algoritmo di Tarjan (in realtà la variante descritta nella Sezione 2.3.3), determineremo il ruolo delle componenti e, di conseguenza, sarà possibile assegnare un ruolo anche ai vertici.

Per automatizzare tutte le idee esposte nei paragrafi precedenti abbiamo prodotto una libreria Java nella quale vengono implementati i concetti esposti nel Capitolo 2. Questa libreria non è orientata ad essere utilizzata come un programma a se stante, bensì mira ad un utilizzo programmatico ed estendibile. Per questo motivo abbiamo ritenuto opportuno riportare nella Sezione 5.2 una descrizione di ogni *pacchetto* in modo da favorirne la comprensione e l'utilizzo.

Nella libreria è presente anche una maschera realizzata utilizzando il framework *SWING* per la renderizzazione di un particolare insieme di risultati, relativi alla composizione delle componenti fortemente connesse: questo è l'unico oggetto che non è possibile utilizzare programmaticamente.

1.5 SINOPSI

In questa sezione elenchiamo i capitoli in cui questo documento è suddiviso e, per ognuno di essi, daremo una breve descrizione del contenuto.

Questo capitolo contiene un'introduzione al lavoro svolto.

Il Capitolo 2 modella da un punto di vista teorico i concetti e gli algoritmi alla base delle implementazioni, in particolare cosa significa visitare un grafo, approfondiremo la strategia *Depth First* e studieremo tre algoritmi per la ricerca delle componenti fortemente connesse.

Il Capitolo 3 formalizza gli obiettivi che vogliamo raggiungere con lo sviluppo della libreria Java. Vengono proposti i casi d'uso che abbiamo implementato e, per ognuno di essi, vengono riportati degli esempi del “prodotto finito” in

modo da visualizzare i risultati se non si è avuto modo di utilizzare la libreria. Inoltre si analizza l'architettura che caratterizza l'intera implementazione.

Il Capitolo 4 termina il lavoro esponendo i risultati ottenuti ed elencando una lista di sviluppi futuri del progetto.

Il Capitolo 5 affronta tutte le questioni inerenti alla fase di implementazione. Si discute la metodologia di sviluppo utilizzata ed i *pacchetti* che compongono il progetto.

1.6 STRUMENTI UTILIZZATI

Per lo sviluppo di questo progetto sono stati utilizzati i seguenti strumenti:

- per la stesura di questo documento è stato usato il motore di formattazione \TeX , utilizzando *Emacs* come editor dei file sorgenti, installando la *major mode AucTex*;
- l'implementazione della libreria Java è stata sviluppata completamente con *Eclipse* versione *Indigo*. La JVM utilizzata per lo sviluppo è quella fornita da *openjdk-6* (versione 24);
- i sorgenti, sia dell'elaborato testuale sia dell'implementazione, sono stati versionati utilizzando il sistema *Git*. Abbiamo utilizzato *Github* come fornitore del servizio. I sorgenti di questo documento possono essere scaricati dall'URL:
<https://github.com/massimo-nocentini/my-undergraduate-thesis>
mentre i sorgenti dell'implementazione Java dall'URL:
<https://github.com/massimo-nocentini/my-undergraduatethesis-java>

ALGORITMI

In questo capitolo affronteremo gli algoritmi alla base delle implementazioni da un punto di vista teorico. Vedremo la tecnica di visita del grafo che abbiamo adottato e diverse soluzioni per il calcolo delle componenti fortemente connesse.

2.1 ALCUNE DEFINIZIONI

Un grafo G è una coppia (V, E) dove V è un insieme di vertici ed E è un insieme di archi. Più precisamente E è una relazione binaria tale che $E \subseteq V \times V$.

Un grafo G si dice *non orientato* se E è una relazione simmetrica, altrimenti si dice *orientato*. Sia G un grafo orientato, un cammino da un vertice v_i ad un vertice v_j è una sequenza di vertici $(v_i, v_{i+1}, \dots, v_{j-1}, v_j)$ tale che $\forall k \in [i, j-1]$ si ha $(v_k, v_{k+1}) \in E$. Tale cammino lo indicheremo con la scrittura $v_i \rightarrow^* v_j$. Nel caso di un grafo non orientato indichiamo un cammino con la scrittura $v_i -^* v_j$.

Un cammino $\pi = v_i \rightarrow^* v_i$ è chiamato un cammino *chiuso*. In particolare, un cammino chiuso è un *ciclo* quando tutti gli archi che lo compongono sono distinti e v_i è l'unico vertice che appare esattamente due volte in π . Due cicli che sono permutazioni l'uno dell'altro sono indistinguibili e, pertanto, si considerano essere lo stesso ciclo.

Un grafo non orientato è *connesso* se $\forall v, w \in V$ esiste un cammino π tale che $\pi = v -^* w$. Un grafo orientato è *fortemente connesso* se $\forall v, w \in V$ esistono due cammini π_1, π_2 tali che $\pi_1 = v \rightarrow^* w \wedge \pi_2 = w \rightarrow^* v$.

Un grafo orientato T è un *albero* se valgono le seguenti proprietà:

- la versione non orientata di T è connessa;
- esiste esattamente un vertice v tale che $\forall w \in V : (w, v) \notin E$. Il vertice v viene chiamato *radice* di T ;
- tutti i vertici distinti dal vertice *radice* hanno esattamente un arco entrante, ovvero $\forall w \neq \text{radice}(T), \exists! r : (r, w) \in E$.

Un insieme di alberi viene definito *foresta*.

Sia G un grafo orientato. Possiamo definire una relazione d'equivalenza sull'insieme di vertici come segue: due vertici v e w sono equivalenti se esiste un cammino chiuso $\pi = v \rightarrow^* v$ che contiene w . Chiamiamo V_i le classi d'equivalenza di questa relazione e sia G_i un sotto grafo di G tale che $G_i = (V_i, E_i)$, con $E_i = \{(r, s) \in E : r, s \in V_i\}$. Allora:

- ogni G_i è fortemente connesso;
- nessun G_i è un sotto grafo proprio di un sotto grafo connesso di G .

I sotto grafi G_i sono chiamati *componenti fortemente connesse* di G .

2.2 VISITARE UN GRAFO

La visita di un grafo G è una procedura fondamentale che permette di estrarre informazioni sulla struttura di G . Per questo motivo è importante avere implementazioni che richiedono tempo polinomiale per terminare la computazione.

2.2.1 Il concetto di visita

Supponiamo che siano dati un grafo $G = (V, E)$ e due vertici $v, w \in V$. Per capire il concetto di visita poniamoci la seguente domanda: come possiamo determinare se esiste un cammino $v \rightarrow^* w$?

Una risposta è quella di costruire un insieme W di vertici raggiungibili da v utilizzando una sequenza di archi in E , e rispondere sì se e solo se $w \in W$. Il processo di costruzione dell'insieme W viene definito *visita* del grafo a partire dal vertice v e diremo *visitato* ogni vertice $s \in W$.

Osserviamo che la visita di un grafo è un'operazione ben diversa dall'iterare sui suoi vertici, in quanto:

- iterando sull'insieme V si raggiunge esattamente ogni vertice per definizione, mentre visitando il grafo si raggiunge un sotto insieme $V' \subseteq V$;
- l'ordine seguito iterando sull'insieme V non è dipendente dalla relazione E del grafo, bensì può essere stabilito a priori. Invece l'ordine con cui vengono raggiunti i vertici in una visita è un fattore che caratterizza una visita da un'altra.

Riassumendo quanto detto in sopra, la visita di un grafo G a partire da un vertice v produce una permutazione π di vertici $(v_{\pi_1}, \dots, v_{\pi_s})$, ognuno di essi raggiungibile da v , di cui è interessante l'ordine codificato nella permutazione, ovvero l'ordine in cui i vertici vengono visitati.

2.2.2 Visita in profondità

Nella precedente sezione abbiamo dato l'idea fondamentale di cosa significa visitare un grafo. Vi sono molte strategie che differenziano due visite, caratterizzate dalla modalità con cui si sceglie il prossimo arco da percorrere.

Quella che differenzia la *visita in profondità*, o *Depth-First Search*, è la seguente:

Nella selezione del prossimo arco da percorrere, si sceglie un arco non ancora percorso uscente dal vertice più recentemente raggiunto.

Possiamo mantenere l'insieme dei vertici *più recentemente raggiunti* in una *pila*.

Per capire meglio il pattern di questa tecnica supponiamo di iniziare la visita dal vertice v . Prima v viene visitato, poi si sceglie un arco non ancora percorso $(v, v_1) \in E$, raggiungendo il vertice v_1 . Adesso v_1 viene visitato e si applica al vicinato di v_1 quanto fatto al vicinato di v . Solo quando tutti i vertici nel vicinato di v_1 vengono visitati è possibile visitare il secondo vicino v_2 di v (è possibile che v_2 venga visitato durante la visita di v_1 , in questo caso non è necessaria nessuna operazione su v_2).

Il complemento "in profondità" cattura l'idea che la visita procede allontanandosi sempre più dal punto di partenza, spostandosi da un vertice ad un vicino di tale vertice, ad un vicino del vicino di tale vertice e così via, tornando indietro solo quando si raggiunge un vertice il cui vicinato è completamente visitato oppure vuoto.

Papadimitriou, in [7] pagina 83, vede questa visita come un algoritmo per risolvere un labirinto, identificandone le idee principali in forma diversa rispetto a come le abbiamo esposte sopra:

[...] the reachability problem is rather like exploring a labyrinth[...], a careless choice of passage might lead you around in circles or might cause you to return to passages that you previously saw but did not investigate.[...] Everybody knows that all you need to explore a labyrinth is a ball of string and a piece of chalk. The chalk prevents looping, the string always takes you back...

Riportando questa citazione su quanto detto sopra, il concetto di vertice visitato corrisponde al *chalk*, mentre la pila contenente i vertici più recentemente raggiunti corrisponde alla *ball of string*.

2.2.3 Pseudo codice e complessità

Formalizziamo la visita in profondità col seguente pseudo codice:

```

1  procedura DFS( $G = (V, E)$ )
    per ogni  $v$  in  $V$ :
        visitato[ $v$ ] = falso

    per ogni  $v$  in  $V$ :
6   se non visitato[ $v$ ]:
        visita( $v, E$ )

```

```

11  procedura visita(v, E):
      visitato[v] = vero
      previsa(v)
      per ogni arco (v, u) in E:
          se non visitato[u]: visita(u, E)
      postvisita(v)

```

Facciamo una breve analisi della complessità. La prima osservazione da fare è che ogni vertice viene esaminato una e una sola volta grazie al vettore *visitato*. Durante l'esplorazione di un vertice si spende una quantità costante di tempo sia per aggiornare il vettore *visitato*, sia per le invocazioni delle funzioni *previsa* e *postvisita*.

Per quanto riguarda la scansione del vicinato, abbiamo per ogni vertice un impiego di tempo diverso: per questo motivo ci conviene considerare gli archi in modo dipendente da come la relazione *E* viene rappresentata in macchina. Supponendo di codificarla con *liste di adiacenza*, ogni arco verrà visitato una e una sola volta in caso di grafo orientato, altrimenti esattamente due volte in caso di grafo non orientato. Segue che il tempo impiegato dalla visita è $O(|V| + |E|)$, lineare nella lunghezza dell'input e quindi polinomiale come avevamo richiesto ad inizio capitolo.

2.2.4 Punti di estensione

È possibile “personalizzare” lo schema classico della visita in profondità in modo da inserire dei “punti di estensione” che permettano di eseguire del comportamento dedicato all'avvenire di eventi salienti durante l'esecuzione della visita. Nello pseudo codice che abbiamo riportato ne abbiamo inseriti due, rappresentati dalle funzioni *previsa* e *postvisita*.

Facciamo un'osservazione sul loro utilizzo e supponiamo che *G* sia un albero. Se implementiamo la visita usando solo la funzione *previsa* allora il risultato è equivalente ad una visita *prefissa* dell'albero, mentre usando solo la funzione *postvisita* allora il risultato è equivalente ad una visita *postfissa*.

Come vedremo nella Sezione 2.3.1, un altro utilizzo è quello di associare ad ogni vertice *v* una coppia di interi che indicano l'intervallo di tempo durante il quale *v* rimane nella pila dei vertici raggiunti.

2.2.5 Costruzione dell'albero DFS

Un risultato della visita in profondità di un grafo non orientato $G = (V, E)$, parallelo alla verifica dell'esistenza di un cammino tra due vertici, è la costruzione di un albero ricoprente di *G*. Per *albero ricoprente* si intende un sotto grafo *T* di *G* tale che $T = (V, E')$, con $E' \subseteq E$, e che *T* sia un albero.

È possibile costruire l'insieme E' aggiungendo gli archi $(v, u) \in E$ tali che $\text{visitato}[u] = \text{falso}$ quando l'arco viene analizzato nella procedura di visita.

Vedremo nel Capitolo 3 come la costruzione di tali alberi verrà raggiunta da una specifica funzionalità del nostro progetto.

2.3 RICERCA DELLE COMPONENTI FORTEMENTE CONNESSE

In questa sezione faremo una panoramica degli algoritmi più conosciuti in letteratura per la ricerca di componenti fortemente connesse di un grafo orientato. Ognuno dei seguenti algoritmi è in realtà una variante della visita in profondità: attraverso piccole modifiche e l'utilizzo dei punti di estensione cattura una idea che lo caratterizza.

2.3.1 Kosaraju: componenti pozzo e inversione del grafo

Quest'idea non introduce nessuna modifica allo schema che abbiamo visto, bensì utilizza la procedura *DFS* due volte su grafi in input differenti.

Un'osservazione che possiamo fare sulla procedura visita, invocata sul vertice u , è che questa termina quando tutti i vertici raggiungibili da u sono stati esplorati. Questo ci permette di ragionare come segue:

Supponiamo di conoscere la decomposizione in componenti fortemente connesse di un grafo. Sia C_i una di queste, tale che sia un pozzo (ovvero che non abbia archi uscenti nel *meta grafo* che ha come vertici le componenti fortemente connesse di cui stiamo ipotizzando l'esistenza). Se $v \in C_i$ allora $C_i = \text{visita}(v)$ (per comodità supponiamo che visita ritorni l'insieme di vertici raggiungibili da v).

Con la precedente osservazione abbiamo un metodo per trovare una componente fortemente connessa, non sappiamo però come trovare né il vertice v né le altre componenti.

Prima di continuare implementiamo le due funzioni *previsita* e *postvisita* in modo da associare ad ogni vertice, rispettivamente il momento in cui viene raggiunto e il momento in cui il suo vicinato viene completamente visitato. Di questi due momenti siamo interessati al secondo che chiamiamo *post*.

Adesso che abbiamo definito la coppia di momenti possiamo osservare che:

il vertice con associato il più grande valore *post* appartiene ad una componente fortemente connessa sorgente.

È facile individuare tali vertici, in quanto sono i vertici per i quali la procedura *DFS* invoca la procedura visita. Però questo non è sufficiente per usare

la precedente osservazione, in quanto quello che si richiede è un vertice che appartenga ad una componente fortemente connessa pozzo.

Per risolvere questo problema possiamo invertire il grafo in input calcolando $G^R = (V, E^{-1})$, con $E^{-1} = \{(w, v) : (v, w) \in E\}$.

Notiamo che le componenti fortemente connesse non cambiano la loro composizione in G^R . Sia C_i una di queste: per definizione di componente fortemente connessa, per ogni coppia di vertici v_1, v_2 esiste un cammino $c_1 = v_1 \rightarrow^* v_2$ ed un cammino $c_2 = v_2 \rightarrow^* v_1$. Invertendo la direzione degli archi, i due cammini continuano ad esistere (c_1 gioca il ruolo di c_2 e viceversa) e quindi la relazione di connessione forte è chiusa rispetto alla operazione \cdot^R . Il vantaggio di passare a G^R è che le componenti sorgenti di G diventano componenti pozzi di G^R .

Adesso possiamo eseguire una visita in profondità su G^R e, per la seconda osservazione in quota, il vertice con associato il massimo *post* appartiene ad una componente fortemente connessa sorgente in G^R e, di conseguenza, ad una componente fortemente connessa pozzo in G , che è quello che volevamo.

Possiamo quindi invocare una visita in profondità sul grafo di input G , scansionando i vertici in ordine decrescente di *post*. Così facendo, per la prima osservazione in quota, identificheremo le componenti fortemente connesse.

Questo algoritmo lavora in tempo lineare, richiedendo circa il doppio del tempo richiesto dalla visita in profondità.

2.3.2 Tarjan: una pila e la funzione lowlink

La caratteristica di questo algoritmo è quella di modificare lo schema di una visita in profondità, implementando i due punti di estensione ed aggiungendo della logica sia dopo aver terminato l'invocazione ricorsiva su ogni vertice non ancora raggiunto del vicinato, sia nel caso si incontrino vertici già visitati.

Gli strumenti utilizzati da questa variante sono:

FUNZIONE numero ad ogni vertice v associa un intero che rappresenta il momento in cui v viene visitato oppure, da un altro punto di vista, quanti vertici sono già stati visitati prima di v ;

FUNZIONE lowlink ad ogni vertice w associa un vertice v antenato di w , tale che:

$$\text{numero}(v) = \min_{a \in \text{antenati}(w)} \{\text{numero}(a)\}$$

UNA PILA che contiene tutti i vertici che sono stati visitati durante l'algoritmo ma che ancora non sono stati associati ad una componente fortemente connessa.

Inoltre Tarjan classifica gli archi dopo aver applicato una visita in profondità in questo modo:

ARCHI DELL'ALBERO DFS sono tutti gli archi che hanno permesso di raggiungere un vertice non ancora visitato e vengono indicati con \rightarrow ;

ARCHI IN AVANTI questi archi non appartengono a nessun albero DFS e collegano un antenato ad un suo discendente;

ARCHI ALL'INDIETRO questi archi non appartengono a nessun albero DFS e collegano un discendente ad un suo antenato. Vengono indicati con $\cdot \rightarrow$;

ARCHI TRASVERSALI questi archi non appartengono a nessun albero DFS e collegano due vertici appartenenti a sotto alberi differenti (questi sotto alberi posso appartenere sia allo stesso albero DFS, sia a due alberi DFS disgiunti). Questi archi sono indicati con $\cdot \rightarrow$.

Un'osservazione da fare riguardo agli archi trasversali è la seguente: se $e = (v, w) \in E$ è un arco trasversale allora $\text{numero}(w) < \text{numero}(v)$ ovvero, graficamente parlando, e è orientato da destra verso sinistra. Motivare questo non è difficile: supponiamo per assurdo che e non abbia questa direzione, allora durante una visita, o verrebbe inserito in un albero DFS (e quindi sarebbe un arco del primo tipo, una contraddizione) oppure raggiungerebbe un vertice già visitato (e quindi sarebbe un arco in avanti o all'indietro, un'altra contraddizione).

Vediamo adesso le idee principali alla base dell'algoritmo. La prima di queste è la seguente:

Sia C una componente fortemente connessa e siano v, w due vertici tali che $v, w \in C$. Sia F la foresta di alberi DFS generati dall'esecuzione di una visita in profondità. Allora v, w hanno un antenato comune nella foresta F . Inoltre, sia u un antenato comune di v, w tale che:

$$\text{numero}(u) = \min_{a \in (\text{antenati}(v) \cap \text{antenati}(w))} \{\text{numero}(a)\}$$

allora $u \in C$.

Per far vedere la precedente osservazione supponiamo che durante una visita in profondità il vertice v sia visitato prima di w ($\text{numero}(v) < \text{numero}(w)$) e, dato che v, w appartengono alla stessa componente fortemente connessa, esista un cammino chiuso $\pi = v \rightarrow^* v$ che contiene w .

Chiamiamo T_u il più piccolo albero che ha come radice u e che contiene tutti i vertici attraversati dal cammino π . Il sotto albero T_u esiste in quanto il cammino π può passare da un albero T_M ad un albero T_m della foresta F se:

$$\max_{r \in T_m} \{\text{numero}(r)\} < \min_{s \in T_M} \{\text{numero}(s)\}$$

percorrendo *archi trasversali*, mentre non arriverà in alberi con numerazioni maggiori rispetto all'albero di provenienza.

Osserviamo però che se il cammino π , partendo da v , attraversasse più di un albero, non sarebbe in grado di tornare nella componente connessa che contiene w perchè gli archi trasversali non possono orientarsi a destra. Questo prova l'esistenza di T_u e v, w hanno un comune antenato, ma dato che T_u contiene tutti i vertici attraversati da π , u essendo in T_u (è la sua radice) viene attraversato a sua volta da π e, di conseguenza, appartiene alla stessa componente fortemente connessa che contiene v, w .

Adesso possiamo affrontare la seconda idea che ci darà un metodo per identificare le componenti fortemente connesse:

Sia C una componente fortemente connessa. Allora i vertici $v_i \in C$ appartengono ad uno stesso sotto albero nella foresta DFS . Inoltre la radice del sotto albero viene chiamata *testa* della componente fortemente connessa C .

La precedente idea riduce il problema di individuare le componenti fortemente connesse al problema di identificare i vertici *radice* di sotto alberi della foresta. Tarjan propone il seguente criterio per individuare i vertici radice:

Un vertice v è *testa* di una componente fortemente connessa se e solo se $\text{numero}(v) = \text{lowlink}(v)$.

Quanto detto in quota ha bisogno di alcune spiegazioni. La prima cosa che dobbiamo definire è come calcolare $\text{lowlink}(v)$:

$$\begin{aligned} \text{lowlink}(v) = & \min\{\{\text{numero}(v)\} \cup \\ & \{\text{numero}(w) : (\exists r \in V : v \rightarrow^* r \rightarrow w) \\ & \wedge (\exists u \in V : u \rightarrow^* v \wedge u \rightarrow^* w \wedge \\ & u, w \in C, \text{ per qualche componente fortemente connessa } C)\}\} \end{aligned}$$

Può sembrare criptico ma non cattura un concetto difficile. Il valore che la funzione lowlink associa ad un vertice v è il minimo valore associato ad un vertice w appartenente alla stessa componente fortemente connessa di v , attraversando un numero qualsiasi di archi seguito da un arco all'indietro.

Pertanto un vertice v è la *testa* di una componente fortemente connessa solo quando da v non è possibile né arrivare in sotto alberi con numerazioni minori (in quanto i vertici in questi contenuti sono già stati assegnati ad alcune componenti fortemente connesse), né raggiungere un antenato (altrimenti non sarebbe la *testa* della componente fortemente connessa). Quanto detto è quello che esprime $\text{numero}(v) = \text{lowlink}(v)$.

Questo algoritmo lavora in tempo polinomiale in quanto il tempo per calcolare la funzione lowlink è costante ed ogni vertice viene inserito nella pila una e una sola volta. Inoltre, per verificare se un vertice è contenuto nella pila,

possiamo utilizzare un vettore di valori booleani, che rende il costo di questa operazione lineare.

2.3.3 Algoritmo di Tarjan mediante due pile

Anche questa variante personalizza lo schema classico della visita in profondità, utilizzando gli stessi punti di estensione della versione descritta nella Sezione 2.3.2.

Il metodo che descriviamo classifica i vertici in due insiemi:

COMPLETI sono tutti i vertici completamente visitati ed assegnati ad una componente fortemente connessa;

PARZIALI sono tutti i vertici appartenenti ad un cammino relativo ad una esecuzione della visita, di cui alcuni possono essere già visitati, ma non ancora associati ad una componente fortemente connessa.

In base a questa classificazione, possiamo indurre una sulle componenti fortemente connesse, avendo rispettivamente delle componenti *complete*, *parziali* oppure *ignote*. Questa classificazione viene mantenuta in modo esplicito usando un vettore che associa ad ogni vertice un valore booleano.

Lo stesso concetto esiste nella versione di Tarjan, codificato come segue: se $\text{lowlink}(v) < \text{numero}(v) \wedge v \in \text{Pila}$ allora v è parziale, altrimenti è completo.

Un altro concetto in comune con la variante di Tarjan è quello di *rappresentante* di una componente fortemente connessa. Un vertice v si dice *rappresentante* della propria componente fortemente connessa se è il primo vertice della componente fortemente connessa ad esser stato visitato. Il lettore noterà subito che è possibile mappare questo concetto su quello di *testa* utilizzato nella variante di Tarjan. Un'altra osservazione è necessaria:

Quando una componente fortemente connessa diviene *completa*, tutti i vertici in essa contenuti divengono a loro volta *completi* e i vertici che sono dei *rappresentanti* perdono tale ruolo.

Questo implica che se un vertice ha il ruolo di rappresentante allora appartiene ad una componente parziale.

Inoltre i vertici rappresentanti permettono di distinguere l'inizio di una componente fortemente connessa dalla successiva e, come vedremo, vengono numerati in ordine crescente rispetto al momento in cui vengono visitati (anche questo concetto lo si ritrova nella variante di Tarjan dove si utilizza la funzione *numero*).

Questa variante, a regime, mantiene due pile: una per mantenere l'insieme dei

vertici parziali, una per mantenere l'insieme dei vertici rappresentanti. Tutti i vertici che appartengono alla stessa componente fortemente connessa occupano posizioni contigue nella pila dei parziali, mentre il primo di essi sarà inserito anche nella pila dei rappresentanti e vi resterà finché la componente fortemente connessa non verrà creata.

L'algoritmo che implementa questa variante procede seguendo la seguente idea:

Ogni volta che un nuovo vertice v viene raggiunto, si inserisce v nelle due pile ed invochiamo ricorsivamente la procedura per ogni vertice vicino di v . Se invece v è già stato visitato, si estrae dalla pila dei rappresentati finché in testa non appare un vertice w tale che $\text{numero}(w) \leq \text{numero}(v)$.

Inizialmente ogni nuovo vertice raggiunto viene inserito in entrambe le pile in quanto potrebbe essere il rappresentante di una nuova componente fortemente connessa.

Commentiamo brevemente quando si debbono estrarre i vertici dalla pila dei rappresentanti: se, scandendo il vicinato, esiste un vertice w già visitato ma non completo, significa che l'arco attraversato per arrivare in w comporta un ciclo e, quindi, si possono rimuovere dalla pila dei rappresentanti tutti i vertici che sono sopra w . Se invece w fosse completo allora appartiene già ad una componente fortemente connessa, e l'arco attraversato è un arco trasversale, come chiamato nella variante di Tarjan.

Infine, come nella variante di Tarjan, si costruisce una componente fortemente connessa quando un vertice, dopo aver completato la visita del suo vicinato, si trova in testa nella pila dei rappresentanti. La nuova componente fortemente connessa sarà composta da tutti i vertici contenuti nella pila dei parziali, contenuti tra la cima della pila e la posizione del rappresentante compresa. Ognuno di loro viene marcato come completo e si continua l'eventuale computazione.

Anche per questa variante il tempo richiesto è lineare: ogni vertice viene inserito ed estratto al più una volta in ogni pila (il caso degenero è un grafo i cui vertici sono tutti isolati, per cui ogni vertice corrisponde ad una componente fortemente connessa), non aggiungendo asintoticamente niente al tempo richiesto dalla visita in profondità.

STUDIO

In questo capitolo affronteremo le principali forze che abbiamo utilizzato nella fase di progettazione e che hanno permesso di dar forma alle nostre idee.

Descriveremo gli obiettivi, i risultati ottenuti e lo schema architetturale alla base di tutta l'implementazione.

3.1 FUNZIONALITÀ

In questa sezione descriveremo le funzionalità che vogliamo implementare e, per ognuna di esse, mostreremo i risultati della realizzazione.

Inoltre, come il lettore capirà dalle prossime righe, ogni funzionalità può essere messa in corrispondenza con il concetto di filtro. Quest'associazione porterà alla naturale astrazione che costituirà l'architettura della nostra libreria.

3.1.1 *Costruzione del grafo a partire da una codifica SBML*

L'operazione fondamentale a cui siamo interessati è poter manipolare un modello metabolico, implementando un sistema di oggetti che permetta di applicare i concetti espressi nel capitolo precedente.

Questa funzionalità è necessaria per poter applicare le successive, in quanto quest'ultime assumono che il modello SBML che si vuole studiare sia già rappresentato mediante un grafo.

Non riportiamo nessuna rappresentazione grafica dei risultati di questa funzionalità in quanto il prodotto è un insieme di oggetti Java che implementano la struttura grafo. Vedremo nella prossima sezione come sarà possibile rappresentarlo graficamente.

Il commento dell'implementazione di questa funzionalità viene rimandato alla Sezione 5.2.2 dell'appendice.

3.1.2 *Rappresentazione grafica del grafo con nodi bianchi e neri*

Con questa funzionalità si vuole costruire un documento interpretabile dai motori di renderizzazione offerti dalla libreria *graphviz*, rappresentando i nodi sorgenti e nodi pozzi con un cerchio completamente riempito (*nodi neri*), mentre i nodi che non sono né sorgenti né pozzi con un cerchio vuoto (*nodi bianchi*). La Figura 3 riporta il risultato di questa funzionalità.

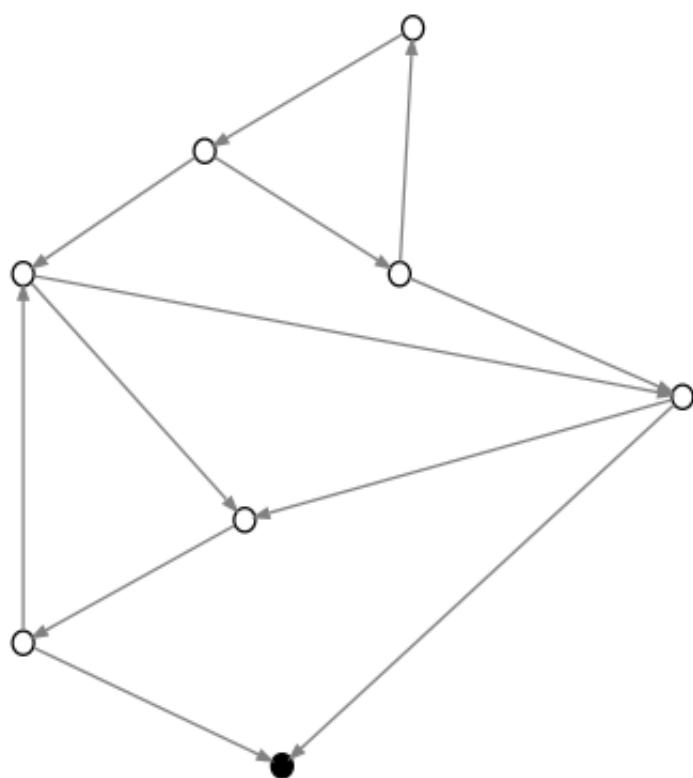


Figura 3: Rappresentazione di un modello SBML mediante un grafo

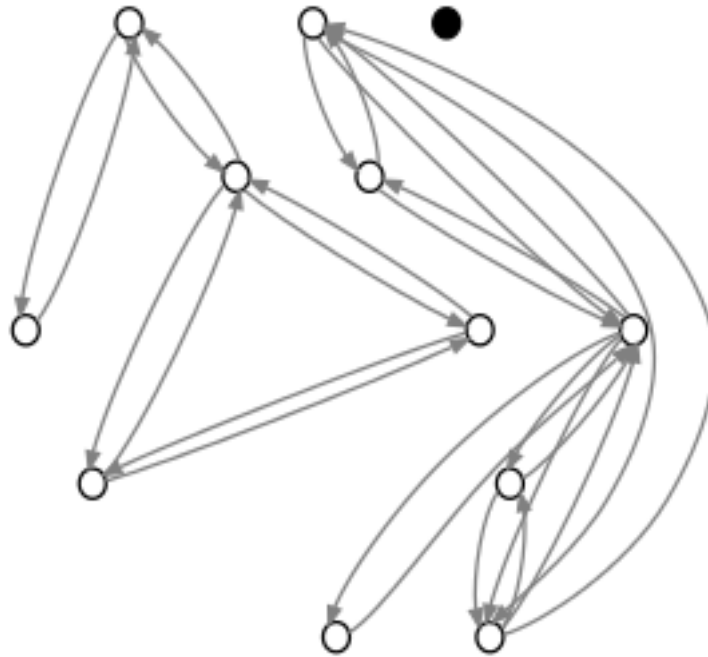


Figura 4: Grafo che si vuole visitare

Il commento dell'implementazione di questa funzionalità viene rimandato alla Sezione 5.2.1 dell'appendice.

3.1.3 Applicare la visita in profondità

Con questa funzionalità si vuole visitare in profondità il grafo in input.

Il risultato prodotto è una foresta di alberi *DFS*, i cui archi sono un sottoinsieme degli archi del grafo originale, attraversati durante la computazione.

Inoltre siamo interessati ad associare, ad ogni vertice v , una coppia di interi che indicano i momenti in cui si visita v e in cui si termina la visita del vicinato di v . La Figura 4 riporta il grafo originale di partenza, preso da [7]: visitandolo in profondità otteniamo la foresta di alberi *DFS* riportata in Figura 5.

Il commento dell'implementazione di questa funzionalità viene rimandato alla Sezioni 5.2.3 e 5.2.5 dell'appendice.

3.1.4 Applicare l'algoritmo di Tarjan

Con questa funzionalità si vuole applicare l'algoritmo di Tarjan per la ricerca delle componenti fortemente connesse al grafo in input.

Il risultato prodotto è un grafo che ha come nodi le componenti fortemente connesse e come archi la relazione di vicinato tra le componenti.

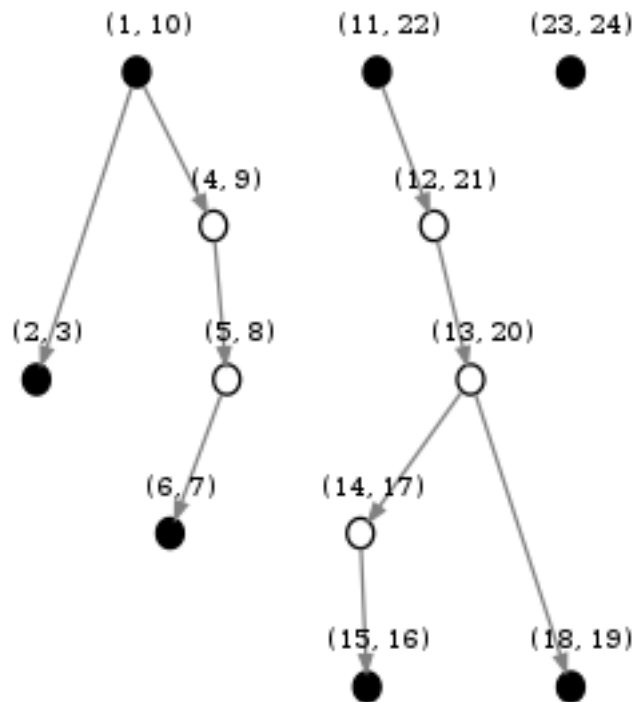


Figura 5: Risultato della visita in profondità del grafo riportato in Figura 4

Inoltre siamo interessati ad associare, per ogni vertice, un intero che indica la cardinalità della componente fortemente connessa che esso rappresenta. La Figura 6 riporta il grafo originale di partenza, preso da [8]: applicando l'algoritmo di Tarjan a tale grafo otteniamo la minimizzazione riportata in Figura 7.

Il commento dell'implementazione di questa funzionalità viene rimandato alle Sezioni 5.2.4 e 5.2.5 dell'appendice.

3.1.5 Rappresentazione tabulare

Con questa funzionalità si vuole rappresentare il grafo in input in formato testuale, usando uno schema tabellare.

Il *side-effect* prodotto è collezionare le informazioni richieste visitando il grafo e scriverle in un file dedicato.

Inoltre, per comodità, vogliamo comporre più di una funzionalità e scrivere nello stesso documento le informazioni relative alle computazioni di ognuna di esse. Vediamo i risultati di una sua esecuzione, allegando un documento prodotto:

```
1 phase-PlainTextStatsPipeFilter-level-2
  NOFVertices: 1
```

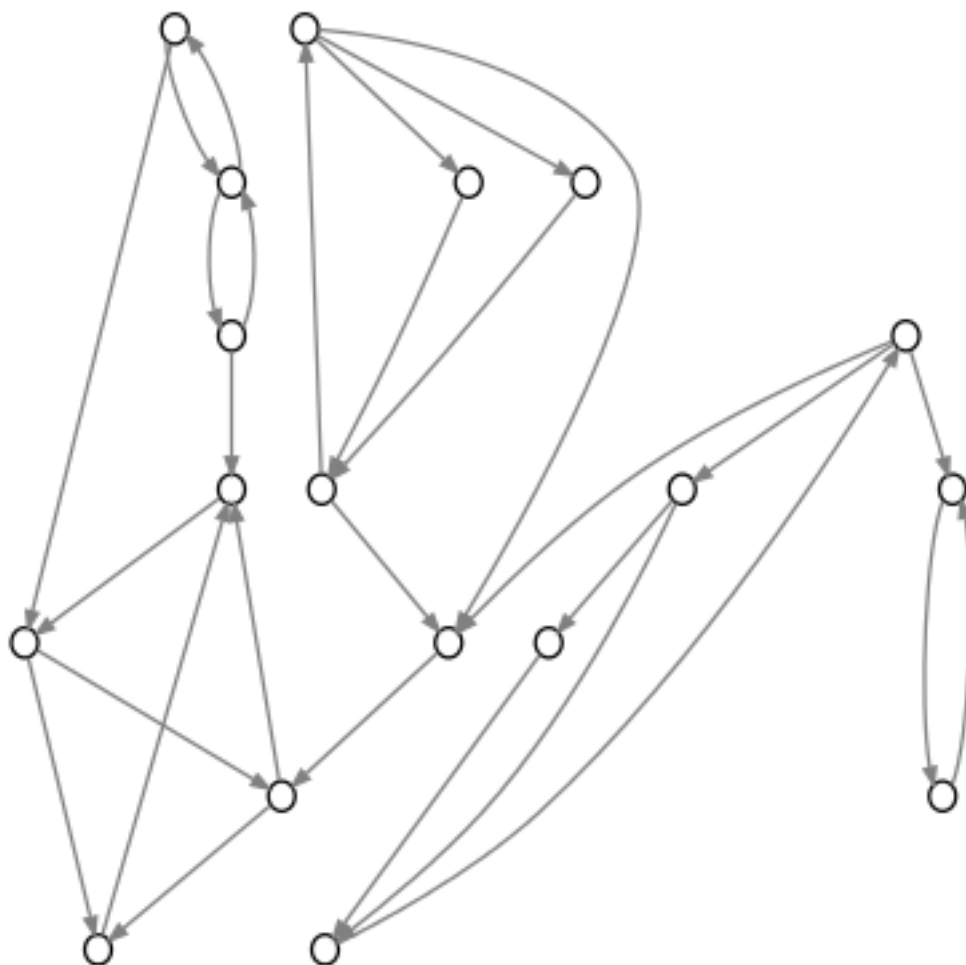


Figura 6: Grafo di cui si vuole ricercare le componenti fortemente connesse

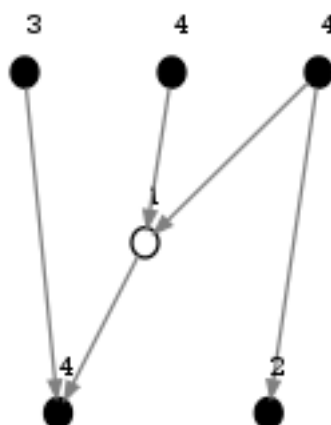


Figura 7: Componenti fortemente connesse del grafo riportato in Figura 6

```

        NOOfComponents: 206
        NOOfEdges: 183
        NOOfSources: 108
6      NOOfSinks: 96
        NOOfWhites: 2

        NOOfVertices: 2
        NOOfComponents: 9
11     NOOfEdges: 1
        NOOfSources: 1
        NOOfSinks: 8
        NOOfWhites: 0

16    NOOfVertices: 3
        NOOfComponents: 5
        NOOfEdges: 0
        NOOfSources: 0
        NOOfSinks: 5
21    NOOfWhites: 0

        NOOfVertices: 4
        NOOfComponents: 1
        NOOfEdges: 0
26    NOOfSources: 0
        NOOfSinks: 1
        NOOfWhites: 0

        NOOfVertices: 396
31    NOOfComponents: 1
        NOOfEdges: 89
        NOOfSources: 0
        NOOfSinks: 0
        NOOfWhites: 1
36

phase—PlainTextStatsPipeFilter—level—0
        NOOfVertices: 639
        NOOfEdges: 2209
        NOOfSources: 108
41    NOOfSinks: 95
        NOOfWhites: 436

```

Questo output necessita di alcune spiegazioni. Nel documento riportato sono collezionate le informazioni sul grafo originale (riportate sotto l'etichetta *phase-*

PlainTextStatsPipeFilter-level-0) e, dopo aver applicato l'algoritmo di Tarjan, quelle sul grafo minimizzato (riportate sotto l'etichetta *phase-PlainTextStatsPipeFilter-level-2*).

Come si nota i formati delle rappresentazioni tabulari sono vicini ma non identici. Dato il seguente blocco relativo al primo passo:

```

2 phase-identifier
  NOfVertices: v
  NOfEdges: e
  NOfSources: s
  NOfSinks: d
  NOfWhites: w

```

allora il grafo in input $G = (V, E)$, al passo *phase-identifier*, è tale che $|E| = e$ e $|V| = v$, di cui s vertici sono sorgenti, d vertici sono pozzi e $w (= v - s - d)$ sono intermedi.

Definiamo adesso la semantica associata alle informazioni collezionate per il secondo passo. Come si può notare, queste informazioni sono composte da un blocco ricorrente per cui ci limiteremo a dare significato solo a questo. Sia dato il blocco:

```

4 NOfVertices: v
  NOfComponents: c
  NOfEdges: e
  NOfSources: s
  NOfSinks: d
  NOfWhites: w

```

allora nel grafo minimizzato esistono c componenti fortemente connesse, ognuna delle quali raggruppa v vertici. Esistono e archi uscenti in totale dalle c componenti e queste sono partizionate in s componenti sorgenti, d componenti pozzo e w componenti intermedie.

Come abbiamo detto ad inizio sezione, questa funzionalità non trasforma il grafo in input, anzi è ortogonale ad un'intera composizione di funzionalità in quanto, elencando i dati raccolti per ogni specifico *phase-identifier*, non ha dipendenze da ognuna di esse.

Il commento dell'implementazione di questa funzionalità viene rimandato alla Sezione 5.2.3 dell'appendice.

3.1.6 Ridurre la complessità collassando i nodi sorgente

Con questa funzionalità si vuole eliminare tutti i vertici sorgenti contenuti nel grafo in input, introducendo al loro posto un nuovo vertice sorgente, tale che il

suo vicinato sia uguale all'unione dei vicinati dei vertici rimossi. Riportiamo il contenuto di un documento generato da questa funzionalità:

```

phase—PlainTextStatsPipeFilter—level—1
  NOfVertices: 386
  NOfEdges: 1339
4 NOfSources: 1
  NOfSinks: 85
  NOfWhites: 300

```

Come si vede dalle informazioni riportate, il nuovo grafo ha un solo vertice sorgente.

Il commento dell'implementazione di questa funzionalità viene rimandato alla Sezione 5.2.3 dell'appendice.

3.1.7 Indagare proprietà di una batteria di modelli

Con questa funzionalità si vogliono rappresentare delle informazioni relative ad una batteria di modelli, dei quali abbiamo calcolato le componenti fortemente connesse, attraverso una semplice interfaccia grafica.

In realtà quello che vogliamo costruire è un visualizzatore di strutture dati, costruite appositamente per contenere le informazioni di interesse, senza essere dipendenti dal risultato di una computazione appena conclusa. Ovvero, vorremmo serializzare tale struttura dati in una path del file system e poterla visualizzare successivamente: secondo noi è un approccio più modulare in quanto non è necessario eseguire ogni volta la computazione per creare la struttura dati, utilizzando la maschera come semplice *render*.

Inoltre siamo interessati a condurre la nostra indagine restringendosi non solo ad un modello (e quindi ad un solo grafo), ma analizzare un insieme di modelli affinché sia possibile studiare in che tipo di componente fortemente connessa ogni metabolito incontrato appare.

La Figura 8 riporta uno screenshot dell'interfaccia, di cui diamo la semantica dei componenti in essa rappresentati, descrivendoli seguendo l'ordine da sinistra verso destra e dall'alto verso il basso:

- nella tabella si elencano nelle righe tutti i modelli che sono stati oggetto di indagine e nelle colonne le tipologie di vertici (*sources*, *whites*, *sinks*): dato un modello e una colonna, si associa una coppia (c, v) . Per esempio, se una coppia (c, v) appartiene alla riga del modello m e alla colonna *whites*, allora nel modello m esistono c componenti fortemente connesse di tipo *whites*, le quali contengono in totale m vertici;
- nella *list-box* alla destra della tabella si riportano delle informazioni relative ai metaboliti. Ogni oggetto è un gruppo di metaboliti in base alla

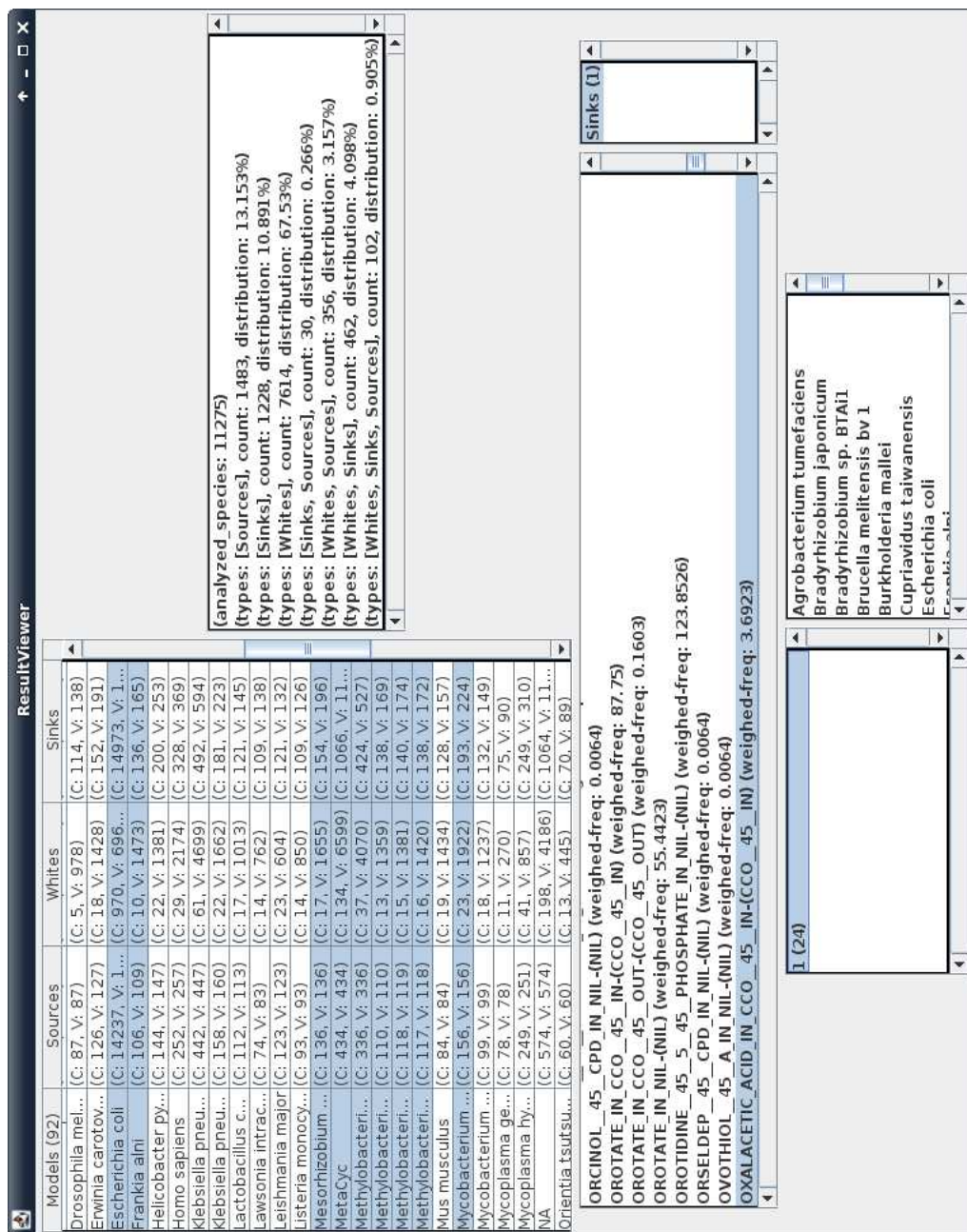


Figura 8: Visualizzatore dell'indagine relativa ad un insieme di modelli

loro tipologia: i gruppi che abbiamo considerato corrispondono alle combinazioni dell'insieme $\{\text{sources}, \text{whites}, \text{sinks}\}$, escludendo l'insieme vuoto. Per ogni gruppo si calcola la rispettiva frequenza in percentuale per avere un raffronto sulla predominanza delle tipologie;

- nella *list-box* sotto la tabella più a sinistra, vengono elencati i metaboliti presenti in almeno un modello oggetto dell'indagine. Ogni metabolito è codificato con una stringa composta dal suo identificatore;
- cliccando su un metabolito s , nella successiva *list-box* vengono elencati i tipi delle componenti fortemente connesse che contengono il metabolito s ;
- cliccando su una tipologia t di componente fortemente connessa, nella successiva *list-box* vengono elencate le cardinalità delle componenti fortemente connesse che contengono il metabolito s e sono di tipologia t . Ad ogni cardinalità si concatena un numero intero che indica la cardinalità dell'insieme rappresentato nell'ultima *list-box*, che descriveremo nel prossimo punto;
- cliccando su una cardinalità c , nell'ultima *list-box* vengono elencati i modelli, la cui applicazione dell'algoritmo di Tarjan, ha prodotto un grafo in cui esiste una componente fortemente connessa tale che contiene il metabolito s , è di tipo t ed ha una cardinalità c .

Non solo le ultime quattro *list-box* sono contestualmente correlate, ma lo sono anche la tabella e la *list-box* con i gruppi per tipologia di vertice. In particolare, selezionando nella tabella una o più modelli, nella *list-box* si aggiornano le informazioni filtrandole rispetto ai metaboliti contenuti nei modelli selezionati. Se la selezione dei modelli è vuota, allora le informazioni sono relative a tutti i modelli.

Abbiamo esteso queste correlazioni anche tra il percorso selezionato nelle ultime quattro *list-box* e la tabella (e di conseguenza alla prima *list-box*). Selezionando un metabolito dalla *list-box* sotto la tabella più a sinistra, vengono selezionati in automatico nella tabella i modelli che lo contengono e, come spiegato nel paragrafo precedente, anche la *list-box* contenente le informazioni per tipologia di vertice verrà aggiornata di conseguenza. Se si raffina il percorso, ad esempio scegliendo la tipologia di vertice e la cardinalità nelle rispettive *list-box*, viene raffinata la selezione nella tabella dei modelli.

Come ultima relazione, selezionando un gruppo nella *list-box* alla destra della tabella si ricercano e selezionano i metaboliti appartenenti al gruppo selezionato.

Il commento dell'implementazione di questa funzionalità viene rimandato alla Sezione 5.2.4 dell'appendice.

3.1.8 Analizzare un grafo non associato ad un modello SBML

Nelle precedenti sezioni abbiamo sempre assunto di lavorare su un modello metabolico, ma niente ci vieta di poter utilizzare la libreria su un grafo arbitrario che costruiamo utilizzando gli oggetti del nostro modello dati.

Questo rende la libreria non strettamente legata ai concetti nel campo della biologia, in particolare delle reti metaboliche, restando aperta ad utilizzi nel campo della teoria dei grafi (in realtà il grafo riportato in Figura 3 non è altro che il grafo utilizzato da Tarjan nel suo articolo [11]).

Il commento dell'implementazione di questa funzionalità viene rimandato alla Sezione 5.2.3 dell'appendice.

3.2 L'ARCHITETTURA *pipes and filters*

In questa sezione descriveremo l'architettura che vogliamo dare al nostro lavoro, coscienti degli obiettivi espressi nelle precedenti sezioni e delle loro caratteristiche.

Daremo prima alcune idee prese dagli articoli che hanno introdotto quest'architettura e, nella seconda parte della sezione, vedremo come possiamo "metterle a punto" nel nostro progetto.

3.2.1 Cenni teorici

Quest'architettura è molto usata nella pratica e vi è molta letteratura da cui poter attingere, pensiamo però che tornare alle sorgenti che per prime hanno permesso la sua diffusione sia di notevole importanza prima di analizzare le varianti e gli studi successivi.

I volumi da cui ho studiato sono quello del Buschmann [9] e dalla miscellanea di Coplien e Schmidt [10].

Schema classico

Nella testata che descrive il pattern in [9], pagina 53, compare:

The *Pipes and Filters* architectural pattern provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allows you to build families or related systems.

L'idea alla base di questo pattern è il principio *separation of concerns*. Quello che si vuol fare è dividere tutte le responsabilità del sistema in sotto problemi, ognuno dei quali può essere implementato con una componente software tale

che *non abbia dipendenze* comportamentali dalle altre (non è client di nessun'altra, non vi è delega di comportamento) e che *coopera* usando l'output di una componente come input per un'altra.

È interessante quello che aggiungono Coplien e Schmidt in [10], pagina 431:

Usually, transformation of input data is done locally and incrementally, so that output may begin before the input is completely read. This means that a filter may start to work as soon as its predecessor produces its first result.

Quest'idea è molto importante se vogliamo introdurre un grado maggiore di parallelismo nel sistema oppure distribuire la computazione. Nel nostro lavoro questo non è stato possibile farlo, in quanto le computazioni che abbiamo incapsulato nei nostri filtri non permettono di produrre un output che possa ritenersi un semilavorato, pronto per essere raffinato dal filtro successivo.

Conseguenze sull'uso dei filtri

Riportiamo ancora un'osservazione fatta in [10], pagina 431:

To preserve the independence of the framework's components, filters may not share a state. The only way to put the results of different filters together is to organize some of them into a sequence such that certain filters perform further transformations on the outputs of others. In addition, a filter should not know the identity of the filters preceding or following it in the computation sequence.

La funzionalità 3.1.4 è in un certo senso quello che Coplien identifica nella frase "*...certain filters perform further transformations...*" per la funzionalità 3.1.7, in quanto quest'ultima assume di lavorare su un grafo minimizzato utilizzando un algoritmo per la ricerca delle componenti fortemente connesse.

Aver introdotto formalmente il concetto di filtro come astrazione è stato molto vantaggioso in quanto ogni computazione, sia effettiva che di adattamento, può essere trattata indipendentemente e in modo trasparente alla stregua delle altre.

Conseguenze sull'uso delle pipes

Riporto ancora da Coplien, pagina 432:

Two principally different possibilities exist for the realization of pipes: pipes may simply be links between filters (such as message calls) or they may be separate components (such as data repositories or sensors). A pipe's only responsibility is to transmit data between filters, eventually by converting their format from the one produced by their sender to the one required by their receiver.

Nel nostro lavoro non abbiamo avuto bisogno della complessità di modellare veri e propri oggetti *pipe* in quanto, come vedremo nella prossima sezione, non è necessario adattare il formato dell'input per coppie diverse di filtri, responsabilità di oggetti *pipe*. Abbiamo quindi scelto la modalità "*links between filters*".

3.2.2 Implementazione in questo progetto

L'implementazione dell'architettura "*pipes and filters*" in questo progetto ha cercato di prendere spunto da quanto detto nella sezione precedente ma allo stesso tempo ha delle caratteristiche che la distinguono dalla linea introdotta.

Abbiamo introdotto la classe *PipeFilter* per modellare il concetto di *filtro* e, come sua proprietà, un riferimento al filtro successivo, per modellare il concetto di *pipe*.

Inoltre possiamo vedere la *pipeline* come una coda di filtri, dove il primo filtro ad essere inserito nella pipeline è il primo ad essere eseguito. Utilizzando la terminologia usata da Buschmann, la nostra implementazione ricade nello scenario "*pull pipeline*", dove la computazione viene innescata dalla richiesta di un client.

Non vi è bisogno di convertire l'output di un filtro per esser trattato come input per un altro, in quanto ogni filtro lavora con l'astrazione *OurModel*, anche se avremo comportamento diverso in base al tipo di oggetti *Vertex* contenuti all'interno dell'input.

La precedente osservazione potrebbe indurre il lettore a pensare alla nostra architettura non tanto come una pipeline, quanto come un sistema di decoratori (vedi pattern *Decorator* in [1]). Quest'interpretazione non è errata, anzi crediamo che indebolire il requisito di stabilire un ordinamento rigido dei filtri presente nel pattern *pipeline* originale, a favore di maggior dinamicità e trasparenza sul formato di input/output, porti ad una soluzione che sarebbe stato sufficiente implementare con un tipico schema con decoratori.

ESPERIMENTI E CONCLUSIONI

In questo capitolo analizzeremo in modo critico il lavoro che abbiamo svolto, commentando le prove effettuate, i risultati ottenuti ed elencheremo dei possibili sviluppi di quanto fatto.

4.1 RISULTATI

In questa sezione riportiamo le informazioni riguardanti le esperienze che abbiamo effettuato: elencheremo le sorgenti dei modelli d'interesse, come utilizzare la libreria sviluppata per riprodurre le prove ed analizzeremo il nostro processo di costruzione dell'insieme \mathbb{B} .

4.1.1 *Modelli biologici studiati*

Da un punto di vista biologico, la maggior parte dei modelli oggetto dei nostri studi sono riferiti a microorganismi batterici e sono reperibili in [12] e in [13] gratuitamente: quelli che si trovano nel primo riferimento sono stati i primi su cui abbiamo lavorato, mentre i secondi sono quelli su cui abbiamo esercitato in modo massivo le nostre implementazioni.

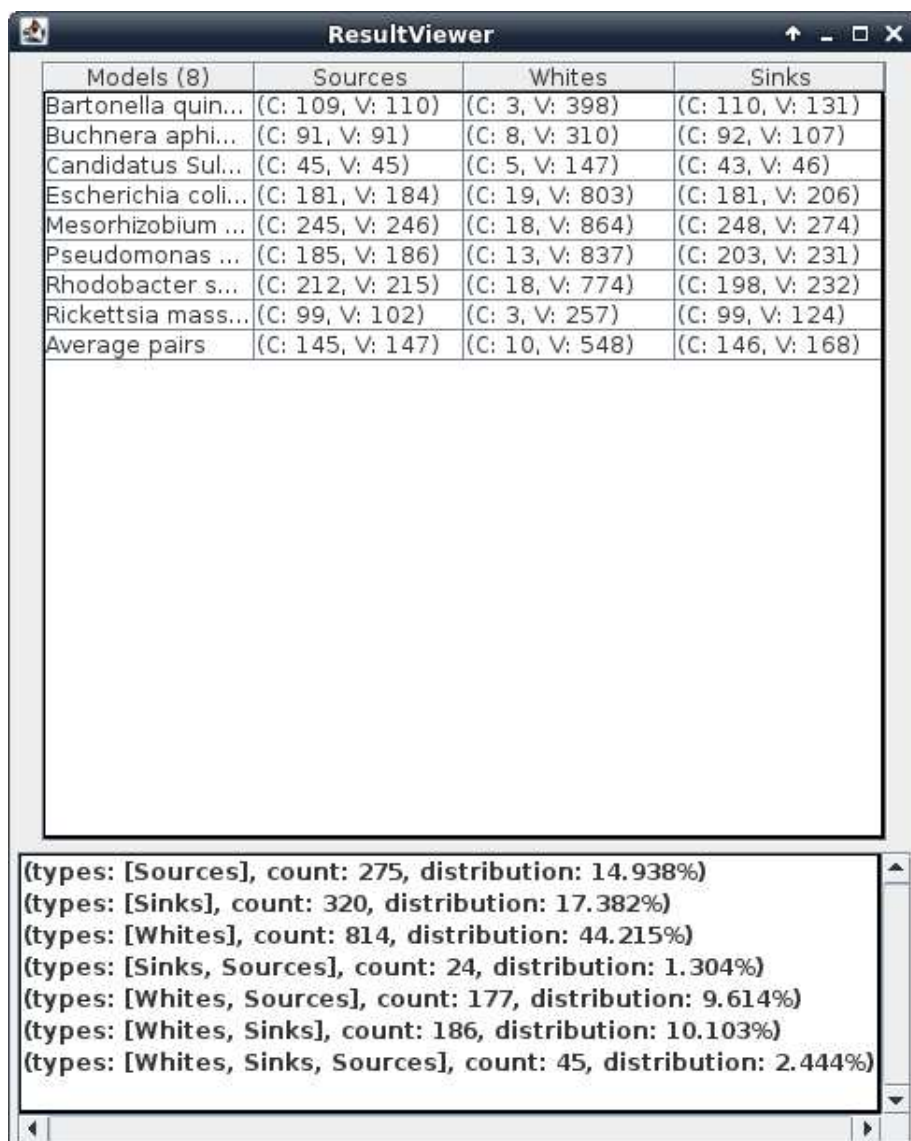
Se si trovassero difficoltà nella loro ricerca, le due sorgenti sono disponibili in [14], come descritto nella Sezione 1.6. Dopo aver scaricato il repository, i modelli relativi a [12] si trovano all'interno della cartella *sbml-test-files*, mentre quelli relativi a [13] nell'archivio *sbml-test-files/BioCyc15.o.tar.gz*.

Nelle Figure 9 e 10 riportiamo le elaborazioni riferite, rispettivamente, ai modelli in [12] e [13]: per i primi sono state analizzati 8 modelli contenenti 1841 metaboliti, per i secondi 165 modelli contenenti 12399 metaboliti.

4.1.2 *Come riprodurre le nostre esperienze*

Ogni prova che abbiamo effettuato è codificata in un *metodo di test*, indipendente dalle altre, per cui è sufficiente compilare i sorgenti reperibili in [14], sia delle classi che dei metodi di test, ed esercitare con il *TestRunner* che si preferisce tutta la batteria di test. I risultati vengono salvati nella cartella *dot-test-files/tests-output*, all'interno troviamo:

- i documenti dot interpretabili con *graphviz* e le relative rappresentazioni sotto forma di immagine *svg*, risultato di quanto descritto nelle Sezioni 3.1.2, 3.1.3 e 3.1.4;



The screenshot shows a window titled "ResultViewer" with a table of models and their statistics. The table has four columns: Models (8), Sources, Whites, and Sinks. Below the table is a large empty box, and at the bottom is a summary section with statistics for various types.

Models (8)	Sources	Whites	Sinks
Bartonella quin...	(C: 109, V: 110)	(C: 3, V: 398)	(C: 110, V: 131)
Buchnera aphi...	(C: 91, V: 91)	(C: 8, V: 310)	(C: 92, V: 107)
Candidatus Sul...	(C: 45, V: 45)	(C: 5, V: 147)	(C: 43, V: 46)
Escherichia coli...	(C: 181, V: 184)	(C: 19, V: 803)	(C: 181, V: 206)
Mesorhizobium ...	(C: 245, V: 246)	(C: 18, V: 864)	(C: 248, V: 274)
Pseudomonas ...	(C: 185, V: 186)	(C: 13, V: 837)	(C: 203, V: 231)
Rhodobacter s...	(C: 212, V: 215)	(C: 18, V: 774)	(C: 198, V: 232)
Rickettsia mass...	(C: 99, V: 102)	(C: 3, V: 257)	(C: 99, V: 124)
Average pairs	(C: 145, V: 147)	(C: 10, V: 548)	(C: 146, V: 168)

(types: [Sources], count: 275, distribution: 14.938%)
 (types: [Sinks], count: 320, distribution: 17.382%)
 (types: [Whites], count: 814, distribution: 44.215%)
 (types: [Sinks, Sources], count: 24, distribution: 1.304%)
 (types: [Whites, Sources], count: 177, distribution: 9.614%)
 (types: [Whites, Sinks], count: 186, distribution: 10.103%)
 (types: [Whites, Sinks, Sources], count: 45, distribution: 2.444%)

Figura 9: Panoramica sui modelli in [12]

ResultViewer			
Models (165)	Sources	Whites	Sinks
Acyrtosiphon ...	(C: 103, V: 105)	(C: 6, V: 1216)	(C: 126, V: 142)
Agrobacterium ...	(C: 134, V: 135)	(C: 16, V: 1463)	(C: 156, V: 201)
Arabidopsis tha...	(C: 101, V: 101)	(C: 37, V: 1628)	(C: 247, V: 269)
Bacillus amyloli...	(C: 96, V: 99)	(C: 14, V: 1126)	(C: 121, V: 139)
Bacillus anthra...	(C: 115, V: 115)	(C: 16, V: 1370)	(C: 142, V: 163)
Bacillus subtilis...	(C: 98, V: 99)	(C: 18, V: 1182)	(C: 119, V: 149)
Bacillus thuring...	(C: 117, V: 120)	(C: 14, V: 1379)	(C: 139, V: 162)
Bartonella bacil...	(C: 74, V: 74)	(C: 11, V: 676)	(C: 98, V: 129)
Bartonella hen...	(C: 83, V: 83)	(C: 11, V: 741)	(C: 111, V: 143)
Bartonella quin...	(C: 81, V: 81)	(C: 10, V: 745)	(C: 110, V: 140)
Bartonella tribo...	(C: 81, V: 81)	(C: 11, V: 776)	(C: 110, V: 140)
Baumannia cic...	(C: 80, V: 81)	(C: 7, V: 529)	(C: 96, V: 119)
Blattabacteriu...	(C: 65, V: 65)	(C: 5, V: 587)	(C: 81, V: 96)
Bordetella tryph...	(C: 65, V: 65)	(C: 13, V: 550)	(C: 82, V: 94)
Bradyrhizobium ...	(C: 145, V: 146)	(C: 15, V: 1620)	(C: 157, V: 192)
Bradyrhizobium ...	(C: 138, V: 139)	(C: 16, V: 1571)	(C: 145, V: 178)
Brucella melite...	(C: 109, V: 109)	(C: 16, V: 1350)	(C: 120, V: 144)
Buchnera aphi...	(C: 135, V: 135)	(C: 10, V: 1484)	(C: 169, V: 195)
Buchnera aphi...	(C: 72, V: 72)	(C: 9, V: 511)	(C: 88, V: 112)
Buchnera aphi...	(C: 96, V: 96)	(C: 12, V: 502)	(C: 102, V: 117)
Buchnera aphi...	(C: 71, V: 71)	(C: 9, V: 493)	(C: 90, V: 118)
Buchnera aphi...	(C: 64, V: 64)	(C: 8, V: 471)	(C: 83, V: 110)
Buchnera aphi...	(C: 49, V: 49)	(C: 8, V: 366)	(C: 65, V: 84)
Burkholderia m...	(C: 124, V: 124)	(C: 15, V: 1509)	(C: 159, V: 202)
Campylobacter ...	(C: 74, V: 77)	(C: 17, V: 841)	(C: 113, V: 136)
(types: [Sources], count: 1586, distribution: 12.791%) (types: [Sinks], count: 1448, distribution: 11.678%) (types: [Whites], count: 7705, distribution: 62.142%) (types: [Sinks, Sources], count: 38, distribution: 0.306%) (types: [Whites, Sources], count: 538, distribution: 4.339%) (types: [Whites, Sinks], count: 847, distribution: 6.831%) (types: [Whites, Sinks, Sources], count: 237, distribution: 1.911%)			

Figura 10: Panoramica sui modelli in [13]

- i file testuali contenenti informazioni sulla struttura dei grafi riferiti sia a dei modelli “ad hoc” riportati anche in questo documento, sia a reti metaboliche, delle quali non è stato possibile riportare una rappresentazione data la loro grande dimensione. Questi file sono il risultato di quanto descritto nelle Sezioni 3.1.5 e 3.1.6;
- le strutture dati contenenti la composizione delle componenti fortemente connesse, relative ai modelli presenti nelle sorgenti citate nella sezione precedente. Per visualizzarle, usiamo la maschera descritta nella Sezione 3.1.7, eseguendo la classe `util.ResultViewer`.

4.1.3 Particolarità di alcuni modelli

Durante l’esecuzione delle nostre prove abbiamo notato una proprietà che caratterizza alcuni modelli in [13]: i microorganismi non sono in biezione con i file in cui sono codificati, bensì il modello di alcuni di loro si ripete in più di un file con informazioni diverse. Questa frammentazione dei modelli è necessaria per catturare il concetto di *strain*, che possiamo definire come una popolazione di microorganismi discendenti da un altro e che sono leggermente differenti. Per questo motivo abbiamo considerato il modello contenuto in ogni file indipendente dagli altri: la Figura 11 riporta il microorganismo *Escherichia coli* e il suo *strain*.

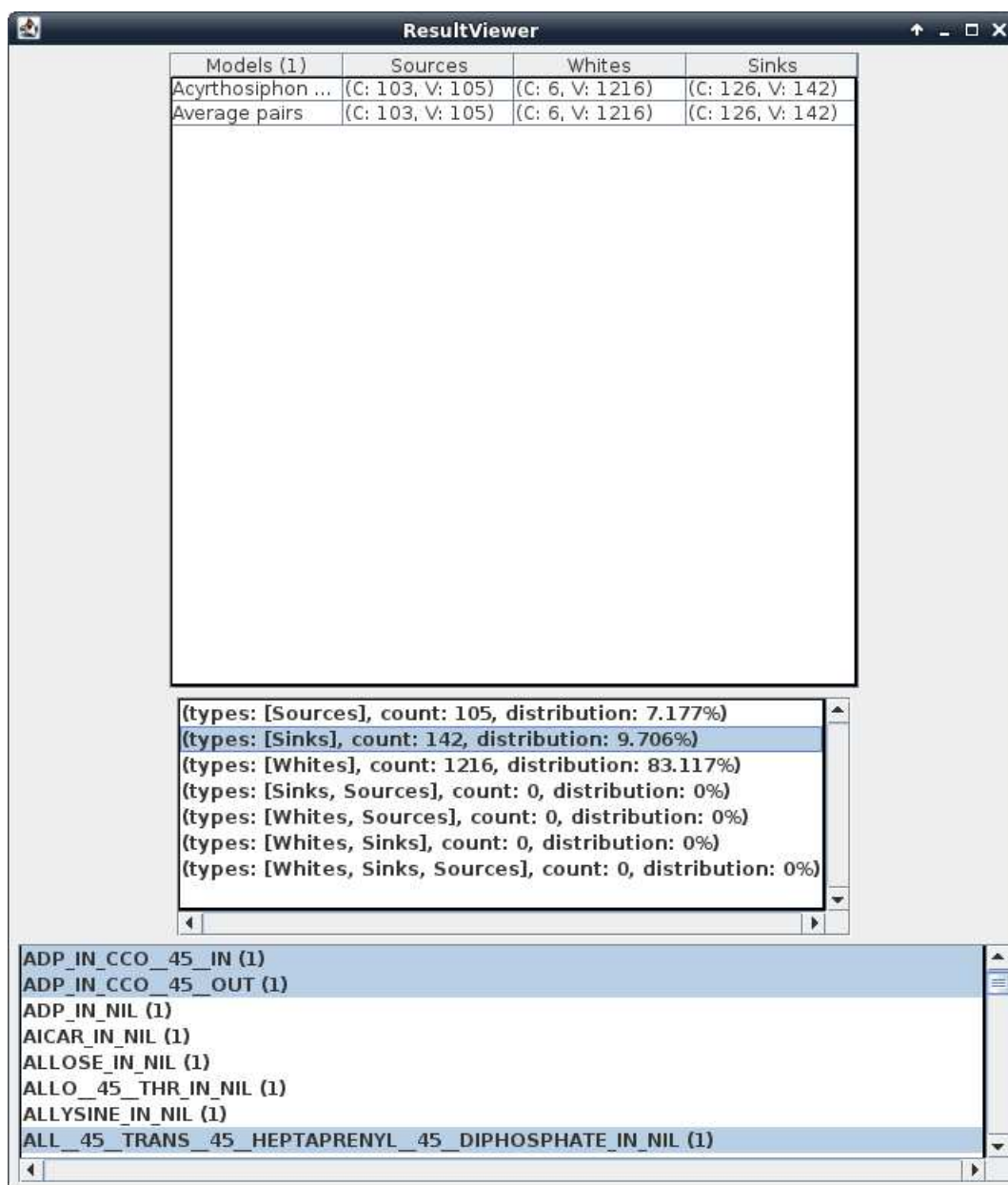
4.1.4 Osservazioni su quanto ottenuto

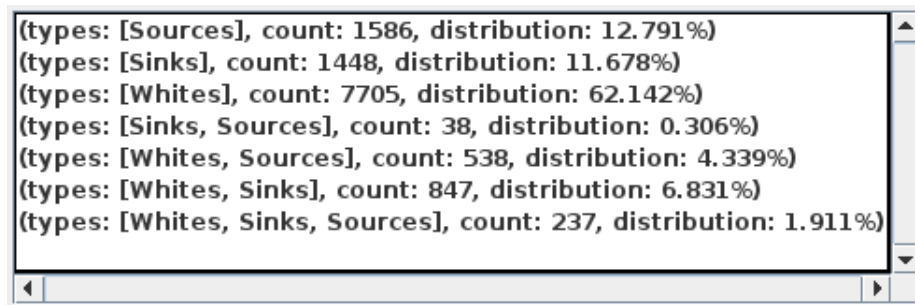
Le osservazioni che possiamo fare sui risultati delle prove possono essere divise in base agli obiettivi che ci poniamo:

- se si vuole costruire l’insieme \mathbb{B} usando una sola rete metabolica allora è possibile assegnare un ruolo ad ogni metabolito: per visualizzare l’assegnazione è sufficiente selezionare nella *list-box* alla destra della tabella il ruolo e, in risposta, il sistema selezionerà, nella rispettiva *list-box*, ogni metabolito a cui è assegnato il ruolo selezionato. Questo è il caso proposto nella Figura 12 che riporta lo studio del modello *Acyrtosiphon*: selezionando la tipologia *pozzo*, osservando la selezione nella *list-box* più in basso, possiamo associare tale ruolo ai metaboliti *ADP_IN_CCO_45_IN*, *ADP_IN_CCO_OUT* e *ALL_45_TRANS_45...* (oltre a tutti quelli non visibili per motivi di spazio) ed includerli nell’insieme \mathbb{B} ;
- se si vuole costruire l’insieme \mathbb{B} usando un insieme di reti metaboliche allora si può avere incoerenza e perdita di precisione. Considerando i modelli in [13] vediamo che esistono dei metaboliti che non hanno un singolo ruolo e questo, seppur con delle percentuali basse, può introdurre errori nella costruzione dell’insieme \mathbb{B} . Questo è il caso proposto nella Figura

ResultViewer			
Models (165)	Sources	Whites	Sinks
Escherichia coli-EC057459	(C: 321, V...	(C: 22, V: 1...	(C: 335, V: ...
Escherichia coli-EC05782	(C: 317, V...	(C: 22, V: 1...	(C: 330, V: ...
Escherichia coli-EC0B1453	(C: 318, V...	(C: 22, V: 1...	(C: 335, V: ...
Escherichia coli-EC0B7450	(C: 315, V...	(C: 22, V: 1...	(C: 331, V: ...
Escherichia coli-EC0BD1038	(C: 133, V...	(C: 22, V: 1...	(C: 154, V: ...
Escherichia coli-EC0BW1039	(C: 315, V...	(C: 21, V: 1...	(C: 330, V: ...
Escherichia coli-EC0DH1037	(C: 313, V...	(C: 21, V: 1...	(C: 327, V: ...
Escherichia coli-EC0ET902	(C: 315, V...	(C: 21, V: 1...	(C: 323, V: ...
Escherichia coli-EC0F1449	(C: 321, V...	(C: 22, V: 1...	(C: 342, V: ...
Escherichia coli-EC0IA121	(C: 319, V...	(C: 22, V: 1...	(C: 331, V: ...
Escherichia coli-ECOLI	(C: 417, V...	(C: 19, V: 1...	(C: 393, V: ...
Escherichia coli-ECOLI149	(C: 317, V...	(C: 22, V: 1...	(C: 332, V: ...
Escherichia coli-ECOLI180	(C: 317, V...	(C: 22, V: 1...	(C: 339, V: ...
Escherichia coli-ECOLI460	(C: 319, V...	(C: 22, V: 1...	(C: 332, V: ...
Escherichia coli-ECOLI83	(C: 316, V...	(C: 21, V: 1...	(C: 342, V: ...
Escherichia coli-ECOS1903	(C: 323, V...	(C: 19, V: 1...	(C: 340, V: ...
Escherichia coli-ECOS8120	(C: 315, V...	(C: 21, V: 1...	(C: 345, V: ...
Escherichia coli-ECOSE461	(C: 319, V...	(C: 20, V: 1...	(C: 340, V: ...
Escherichia coli-ECOUT455	(C: 317, V...	(C: 22, V: 1...	(C: 333, V: ...
Escherichia coli-ECUMN127	(C: 322, V...	(C: 21, V: 1...	(C: 336, V: ...
Escherichia coli-ERYLH300	(C: 317, V...	(C: 22, V: 1...	(C: 331, V: ...
Escherichia coli-ERYLH456	(C: 316, V...	(C: 22, V: 1...	(C: 335, V: ...
Escherichia coli-ESC061044	(C: 323, V...	(C: 22, V: 1...	(C: 335, V: ...
Escherichia coli-ESC401042	(C: 323, V...	(C: 22, V: 1...	(C: 337, V: ...
Escherichia coli-ESC451043	(C: 322, V...	(C: 22, V: 1...	(C: 334, V: ...

Figura 11: Strain relative al microorganismo *Escherichia coli*

Figura 12: Studio del solo modello *Acyrthosiphon*



(types: [Sources], count: 1586, distribution: 12.791%)
(types: [Sinks], count: 1448, distribution: 11.678%)
(types: [Whites], count: 7705, distribution: 62.142%)
(types: [Sinks, Sources], count: 38, distribution: 0.306%)
(types: [Whites, Sources], count: 538, distribution: 4.339%)
(types: [Whites, Sinks], count: 847, distribution: 6.831%)
(types: [Whites, Sinks, Sources], count: 237, distribution: 1.911%)

Figura 13: Indagine su tutti i modelli in [13]

13: vediamo che esistono 38 metaboliti che, in almeno due modelli, hanno rispettivamente il ruolo di pozzo e di sorgente, 538 metaboliti invece hanno quello di sorgente e intermedio, 847 hanno quello di intermedio e pozzo, infine 237 metaboliti che, in almeno tre modelli, hanno tre ruoli diversi. Quest'ultimo caso, insieme alle combinazioni {Sources, Whites} e {Whites, Sinks} (che in totale hanno più del 10% dei metaboliti totali), produce maggior incertezza in quanto non è possibile decidere la loro appartenenza all'insieme \mathbb{B} ;

- se si vuole semplificare la rete utilizzando il meta grafo delle componenti fortemente connesse osserviamo che, per tutti i modelli studiati, non abbiamo un grafo significativo in quanto vi sono molte componenti sorgenti e pozzo, mentre sono molto poche le componenti intermedie. Questo è il caso proposto in Figura 14, nella quale abbiamo evidenziato la riga che riporta le coppie medie calcolate su tutte le coppie precedenti: notiamo che le componenti sorgenti e pozzo non sono composte da molti metaboliti, mentre le componenti intermedie lo sono. Inoltre il meta grafo non è utile per elaborazioni successive per la sua topologia a "clessidra": vi sono molte componenti sorgenti che raggiungono poche componenti intermedie che, a loro volta, raggiungono molte componenti pozzo.

4.2 SVILUPPI FUTURI

I seguenti punti possono essere presi come basi di partenza per sviluppi futuri del progetto:

COMPOSIZIONE DELLE COMPONENTI realizzare un filtro che produca un file contenente l'insieme di metaboliti che compongono ogni componente fortemente connessa. Questo potrebbe risultare utile agli scopi del lavoro in [6], agevolando l'identificazione dei metaboliti tra cui scegliere i rappresentanti dei *seed sets*;

Models (165)	Sources	Whites	Sinks
Salmonella enter...	(C: 132, V: 133)	(C: 20, V: 1417)	(C: 157, V: 189)
Shigella boydii-S...	(C: 316, V: 321)	(C: 23, V: 1524)	(C: 336, V: 374)
Shigella dysente...	(C: 301, V: 305)	(C: 20, V: 1371)	(C: 327, V: 366)
Shigella flexneri...	(C: 316, V: 321)	(C: 22, V: 1545)	(C: 331, V: 369)
Shigella flexneri...	(C: 314, V: 319)	(C: 23, V: 1537)	(C: 331, V: 369)
Shigella flexneri...	(C: 316, V: 321)	(C: 22, V: 1539)	(C: 331, V: 369)
Shigella sonnei...	(C: 322, V: 327)	(C: 23, V: 1480)	(C: 334, V: 372)
Sinorhizobium m...	(C: 121, V: 121)	(C: 21, V: 1500)	(C: 147, V: 180)
Sodalis glossinid...	(C: 96, V: 96)	(C: 17, V: 1137)	(C: 117, V: 144)
Streptococcus a...	(C: 109, V: 112)	(C: 18, V: 903)	(C: 131, V: 162)
Streptococcus t...	(C: 90, V: 93)	(C: 16, V: 810)	(C: 116, V: 145)
Thiomicrospira c...	(C: 79, V: 79)	(C: 13, V: 1021)	(C: 122, V: 160)
Trypanosoma br...	(C: 96, V: 96)	(C: 29, V: 812)	(C: 108, V: 119)
Trypanosoma br...	(C: 75, V: 75)	(C: 3, V: 729)	(C: 84, V: 100)
Trypanosoma br...	(C: 60, V: 60)	(C: 1, V: 440)	(C: 67, V: 78)
Trypanosoma br...	(C: 80, V: 80)	(C: 4, V: 672)	(C: 86, V: 97)
Vibrio cholerae-V...	(C: 107, V: 107)	(C: 10, V: 1172)	(C: 134, V: 169)
Wigglesworthia g...	(C: 65, V: 65)	(C: 11, V: 535)	(C: 76, V: 94)
Wolbachia endo...	(C: 61, V: 61)	(C: 14, V: 619)	(C: 73, V: 91)
Wolbachia pipien...	(C: 71, V: 71)	(C: 14, V: 581)	(C: 75, V: 93)
Wolinella succin...	(C: 78, V: 78)	(C: 12, V: 848)	(C: 105, V: 136)
Xylella fastidiosa...	(C: 67, V: 68)	(C: 9, V: 996)	(C: 103, V: 125)
Yersinia pestis-Y...	(C: 132, V: 135)	(C: 19, V: 1348)	(C: 152, V: 186)
Yersinia pseudot...	(C: 122, V: 122)	(C: 22, V: 1465)	(C: 156, V: 192)
Average pairs	(C: 165, V: 168)	(C: 17, V: 1167)	(C: 191, V: 220)

Figura 14: Numero e cardinalità delle componenti fortemente connesse per i modelli in [13]

DEPENDENCY INJECTION rifattorizzare le parti del codice in cui vengono utilizzati degli oggetti *factory* sostituendoli con motori di *dependency injection*, applicando il principio di *inversion of control*;

DOMAIN SPECIFIC LANGUAGE specificare e implementare un *DSL* per poter descrivere e assemblare la pipeline in modo dichiarativo, senza dover scendere al livello delle interfacce e delle classi che abbiamo implementato. La definizione della pipeline potrebbe essere caricata da riga di comando oppure da un file esterno;

GRAPHVIZ INTERFACE utilizzare una libreria che permette di usare programmaticamente gli oggetti e le implementazioni fornite dalla libreria *graphviz*. Nell'attuale lavoro si utilizzano tali oggetti solo eseguendoli in modalità di comando, equivalente ad una invocazione da terminale. Questo permetterebbe di avere un maggior grado di portabilità del lavoro svolto, eliminando la dipendenza da un'installazione a monte di *graphviz*;

FILE DI CONFIGURAZIONE creare un file di configurazione nel quale impostare il motore di *graphviz* che si vuole utilizzare per la renderizzazione dei grafi (ad esempio esistono anche *neato* e molti altri oltre a quello che abbiamo usato in questo lavoro), il preambolo dei documenti dot nei quali si specifica le formattazioni, i colori e le dimensioni di nodi e archi.

Inoltre potrebbe essere interessante esprimere quanto detto nel precedente paragrafo con uno stesso DSL, in modo da poter codificare le proprie pipeline in batterie e lasciare al codice la responsabilità di costruire i necessari oggetti ed eseguire le rispettive computazioni;

RAPPRESENTARE L'ALBERO DFS NEL GRAFO COMPLETO costruire un nuovo filtro che permetta di comporre la funzionalità per eseguire una visita in profondità e rappresentare l'albero risultante "sopra" il grafo visitato. Per realizzarlo si potrebbe modificare la logica che implementa la visita eseguendola per un vertice scelto (oppure dato) e, successivamente, colorare gli archi del grafo che sono anche archi dell'albero *DFS*, usando un colore più scuro (anche questo passo è già predisposto in quanto il colore grigio usato attualmente non è stato scelto a caso).

APPENDICE

In quest'appendice daremo delle informazioni di alto livello riguardanti l'implementazione dei concetti descritti nel Capitolo 3, descriveremo la metodologia di sviluppo utilizzata e faremo una panoramica sulle funzionalità incapsulate in ogni pacchetto della libreria.

5.1 METODOLOGIA DI SVILUPPO ADOTTATA

La fase di implementazione è stata eseguita utilizzando la metodologia di sviluppo *Test-Driven Development*: abbiamo cercato di studiare ed applicare quanto Beck espone nel volume [2].

Procedere per piccoli passi ed in modo incrementale ci ha permesso di sviluppare un sistema ad oggetti, flessibile e mantenibile, avendo inoltre una *runnable specification* composta dall'insieme delle batterie di test implementate.

Il concetto di *Learning Test* ci è stato particolarmente utile nella fase iniziale del progetto, durante la quale abbiamo familiarizzato con una libreria esterna per il parsing di modelli codificati con il linguaggio SBML. Nella prossima sezione lo approfondiremo.

5.1.1 *Learning tests*

Questa tipologia di test viene introdotta in [2], pagina 136: i test scritti seguendo hanno l'obiettivo di familiarizzare l'utilizzo di una libreria esterna.

Nel nostro caso abbiamo applicato questi concetti per conoscere la libreria *JSBML*, la cui documentazione è estesa e ben dettagliata, ma rimangono degli aspetti che non è possibile studiarli a priori.

Questi test permettono di apprendere il funzionamento di un sistema di oggetti fornito da terzi e proteggono il sistema da noi implementato da eventuali problemi introdotti nei nuovi rilasci della libreria esterna.

Nel nostro caso, se viene fornita una nuova versione della libreria *JSBML*, abbiamo il vantaggio di avere delle asserzioni che verificano i concetti che sono dipendenze per il nostro lavoro: siamo interessati che nessuna di queste sia violata da un nuovo rilascio.

Senza questi test avremmo dovuto provare il software manualmente per controllare che il comportamento non sia cambiato. Avendoli, invece, è sufficiente esercitare di nuovo tutte le batterie per assicurare che i concetti su cui ci siamo basati non abbiano subito modifiche.

Nella prossime sezioni faremo una breve panoramica sui pacchetti che contengono le implementazioni.

5.2 DESCRIZIONE DEI PACCHETTI DEL PROGETTO

Nelle prossime sezioni descriveremo gli oggetti implementati nei pacchetti Java che compongono il progetto. Ogni pacchetto verrà descritto in una sezione dedicata, elencandone i concetti principali ed alcuni dettagli implementativi di interesse.

5.2.1 Pacchetto *dotInterface*

Questo pacchetto incapsula tutte quelle astrazioni necessarie alla creazione di un documento compilabile con i motori messi a disposizione dalla libreria *graphviz*.

Funzionalità implementate

Le funzionalità fornite da questo pacchetto sono le seguenti:

- definire i contratti fondamentali che codificano le responsabilità affinché un oggetto sia esportabile in formato dot e, in modo duale, sia capace di esportare in formato dot un altro oggetto. Questi due contratti sono le astrazioni più importanti di tutto il pacchetto e sono dipendenze delle classi *Vertex* e *OurModel*;
- incapsulare in un'unica classe tutte quelle funzionalità e caratteristiche utilizzate durante la generazione dell'output, alcune di esse dipendenti dal sistema operativo che ospita la *Java Virtual Machine*, ad esempio: il separatore utilizzato nei percorsi di file e directory nel file system, posizioni degli output della libreria e delle cartelle dove trovare i modelli di input;
- fornire un wrapper per oggetti di tipo *Writer*, per concatenare alla rappresentazione testuale dell'oggetto il carattere ';', i due caratteri di *carriage return* e *new line* per appendere il prossimo contenuto su una nuova linea, agevolando la scrittura di documenti dot.

Classi

Procediamo con ordine nel descrivere le principali classi:

DOTEXPORTER Il contratto *DotExporter* definisce quali sono i messaggi che un esportatore deve essere in grado di comprendere per costruire documenti in formato dot, lavorando su un grafo codificato con le astrazioni di *Vertex* e *OurModel*.

I messaggi definiti in *DotExporter* riguardano le componenti basilari di un grafo che si possono rappresentare graficamente: il vertice (con eventuale etichetta esterna al nodo) e l'arco (di cui non è stato necessario introdurre un'etichetta). Riporto il codice per maggior chiarezza:

```

public interface DotExporter extends
    DotDocumentPartHandler {
        DotExporter buildVertexDefinition(Vertex vertex);
        DotExporter collectCompleteContent(Writer
            outputPlugObject);
4      DotExporter buildEdgeDefinition(Vertex source, Vertex
            neighbour);
        DotExporter buildVertexLabelOutsideBoxDefinition(Vertex
            vertex);
    }

9  public interface DotExportable {
        void acceptExporter(DotExporter exporter);
    }

```

Per non accoppiare in modo forte la modalità di salvataggio dell'output nei messaggi dell'interfaccia, ne viene esposto uno che ha come argomento un oggetto di tipo *Writer*, appartenente alla libreria di *io* fornita in *openjdk*.

Questo non limita l'insieme di destinazioni dell'output, bensì lascia aperte molte scelte ed eventuali client del contratto *DotExporter* potranno utilizzare i propri oggetti come destinazione dell'output, potendoli usare per computazioni successive. Nelle nostre implementazioni abbiamo usato quasi sempre oggetti di tipo *FileWriter* come destinazioni concrete.

DOTEXPORTABLE Questo contratto permette di definire quali responsabilità devono essere implementate affinché un oggetto sia esportabile in formato dot.

Non è un'interfaccia molto ricca, ma il solo messaggio *acceptExporter* è sufficiente per catturare la nostra idea. Questa struttura si avvicina molto a quella che viene esposta nel pattern *Visitor* in [1], anche se la nostra implementazione non la ricalca fedelmente: il taglio che abbiamo voluto dare alla coppia *DotExporter* e *DotExportable* usa lo stesso il *double dispatch*, con la differenza di non esporre nel contratto *DotExporter* i metodi relativi alle classi concrete di *Vertex*, avendo un solo messaggio *buildVertexDefinition* come sopra riportato.

5.2.2 Pacchetto *JSBMLInterface*

Questo pacchetto contiene le implementazioni dei concetti che riguardano l'interfacciamento con modelli SBML utilizzando la libreria *JSBML* (vedi [4]).

Funzionalità implementate

Le funzionalità fornite da questo pacchetto sono le seguenti:

- astrarre dalla libreria *JSBML* e dal suo modello dati. In questo modo l'unico contesto del progetto dipendente dalla libreria *JSBML* è confinato a questo singolo pacchetto e gli altri pacchetti non dovranno essere a conoscenza di come viene codificato il modello. Usando questa strategia se si vorrà sostituire la libreria di interfacciamento con modelli SBML, sarà necessario apportare le modifiche solo in questo pacchetto, lasciando tutto il restante codice del progetto inalterato;
- interpretare il modello fornito dalla libreria *JSBML*, costruire gli elementi fondamentali del nostro modello dati e renderlo disponibile per successive computazioni.

Classi

In questo pacchetto la classe principale è *Connector*, la quale incapsula la responsabilità di delegare alla libreria *JSBML* la lettura di un modello SBML e, successivamente, interpretare il risultato della lettura per costruire un nostro modello dati interno.

La parte di interpretazione del modello letto dalla libreria *JSBML* è quella più interessante in quanto elabora e seleziona solo quelle informazioni del modello SBML che effettivamente sono necessarie al nostro lavoro.

Questo procedimento è implementato principalmente nel metodo *readReactions*, il quale costruisce un insieme di vertici a partire da una collezione di reazioni, i quali verranno usati per costruire il modello di dominio.

5.2.3 *Pacchetto Model*

Questo pacchetto contiene le implementazioni dei concetti che riguardano le più importanti e ricche astrazioni del progetto.

Funzionalità implementate

Le funzionalità fornite da questo pacchetto sono le seguenti:

- definire l'astrazione di vertice e delle sue specializzazioni, necessarie per implementare gli altri moduli del progetto. Quest'astrazione può essere modellata a livello teorico con un *abstract data type*, in quanto non vogliamo che i client siano a conoscenza delle specifiche realizzazioni (come invece lo sarebbe se fosse stato un *algebraic data type*), vogliamo invece usare un approccio trasparente per mantenere flessibile il sistema. Le specializzazioni del concetto di vertice che abbiamo implementato sono quelle di

vertice semplice, di vertice da usare nella visita in profondità e negli algoritmi per la ricerca delle componenti fortemente connesse. Per la creazione di questi oggetti si utilizzano degli oggetti *factory* (vedi relativo pattern in [1]);

- implementare l'astrazione di grafo, arricchendola con comportamenti per poterne modificare la struttura ed eseguire delle computazioni su di esso: è possibile usare due idee prese dal paradigma funzionale per agire sui vertici del grafo ed è fornita una implementazione della visita in profondità che espone i punti di estensione come esposto nella Sezione 2.2.4;
- interfacciare le astrazioni di vertice e grafo con gli oggetti definiti nel pacchetto *DotInterface*, per darne una rappresentazione grafica che riporti le idee definite nell'articolo [5].

Classi

Procediamo con ordine nel descrivere le idee principali catturate in ogni contratto:

VERTEX L'interfaccia *Vertex* è l'astrazione principale dell'intera libreria. Tramite *Vertex* possiamo richiedere ad un valido implementatore di gestire le informazioni sul vicinato di un vertice, dando la possibilità sia di aggiungere vicini che predecessori (quindi modelliamo sia una lista di adiacenza, sia una lista di incidenza). Questo ci permette di avere un costo nell'ordine $O(m + n)$ dove m è il numero totale di archi ed n è il numero di nodi presenti nel grafo codificato nel nostro modello. Inoltre, per poter collassare le sorgenti in una unica, è possibile richiedere la "rottura" della relazione di vicinato per poter successivamente cancellare il vertice dal grafo.

Vertex permette di richiedere informazioni sulle caratteristiche del vertice, ad esempio se un vertice è sorgente o pozzo, qual è il suo vicinato, quali sono le informazioni che lo distinguono (riferite alla metabolito e al compartimento recuperate dal modello SBML di origine).

Tramite questo contratto possiamo interfacciare le implementazioni definite nel pacchetto *dotInterface*, richiedendo di accettare un esportatore per costruire un documento in formato *dot*.

È possibile richiedere anche delle informazioni di carattere testuale rappresentandole in una tabella, dalle quali si può capire meglio la struttura del grafo.

Una proprietà che questo contratto richiede è quella di riscrivere i metodi *equals* e *hashCode* in modo che tutti gli oggetti implementatori di questa interfaccia possano essere trattati come *value objects*, non distinguendoli per riferimento in memoria (implementazione di default del metodo *equals* fornito da

openjdk), bensì per le informazioni che questi incapsulano (e quindi poter usare *late-binding* ed avere comportamento polimorfo in base all'oggetto che riceve i suddetti messaggi). Abbiamo effettuato questa scelta in modo da poter creare implementatori di *Vertex* all'occorrenza con le stesse caratteristiche, senza dover gestire una struttura dati per la memorizzazione e la ricerca dell'unico oggetto creato.

Come ultima proprietà, questo contratto impone una relazione d'ordine totale sull'insieme degli implementatori, in modo da rendere gli algoritmi deterministici nel momento di selezione dei vertici.

OURMODEL La classe concreta *OurModel* implementa l'astrazione grafo da utilizzare come modello di dominio per le computazioni descritte nel Capitolo 3.

La prima responsabilità di *OurModel* è costruire il modello in diverse modalità: a partire da un insieme di vertici già costruiti (quindi con le relazioni di vicinato già fissate) oppure a partire da un modello SBML esistente in una path del file system.

La seconda è templetizzare l'algoritmo della visita in profondità, introducendo dei punti di estensione sui quali è possibile "personalizzare" il comportamento della visita ed implementare delle varianti. Quelle che abbiamo implementato in questo lavoro sono quelle definite nel Capitolo 2, ma questo non limita un utilizzatore della libreria di definire il proprio comportamento e di usare l'implementazione della visita per algoritmi che non abbiamo trattato.

IMPLEMENTAZIONI DI VERTEX Abbiamo molti comportamenti diversi che vogliamo poter utilizzare come istanze dell'astrazione definita dal contratto *Vertex*. Il vantaggio di aver implementato il codice riferendosi sempre al contratto e non alle singole caratterizzazioni è di essere trasparente e poter interscambiare i comportamenti specifici, lasciando immutato il codice che implementa i vari algoritmi.

Vediamo quali sono gli implementatori del contratto *Vertex*, descrivendoli brevemente.

La prima implementazione introdotta è stata *SimpleVertex*, che cattura il comportamento di un vertice "normale", ovvero l'implementazione più semplice possibile del contratto. Anche se può sembrare scontata, è il mattone utilizzato più o meno indirettamente dalle implementazioni più complesse: queste sono dei *wrapper* che delegano la maggior parte del comportamento a questa implementazione base, ridefinendo solo i metodi per i tratti che le distinguono.

La seconda implementazione introdotta è stata *DfsWrapperVertex*, che associa l'analogo di un "timestamp" nei momenti in cui la visita in profondità raggiunge un vertice e in cui lo abbandona.

La terza implementazione introdotta è stata *ConnectedComponentWrapperVertex*, in coppia con *TarjanWrapperVertex*. Queste implementano, in modo pura-

mente orientato agli oggetti, l'algoritmo per la ricerca delle componenti fortemente connesse. In particolare *ConnectedComponentWrapperVertex* rappresenta una componente fortemente connessa e mantiene l'insieme dei vertici del grafo di origine di cui è composta, mentre *TarjanWrapperVertex* cattura la responsabilità di mantenere la relazione di vicinato tra le componenti fortemente connesse. Inoltre abbiamo introdotto l'astrazione *ConnectedComponentInfoRecorder* per poter sviluppare un semplice programma per la visualizzazione di semplici statistiche come descritto nel Capitolo 3.

L'implementazione *VertexWithLabelWrapperVertex*, ortogonale alle precedenti, riporta un'etichetta sopra alla rappresentazione di un vertice nella generazione dell'output grafico. Questo comportamento è possibile utilizzarlo in modo trasparente e permette di fattorizzare il codice che cattura la decorazione da quello che cattura le particolari implementazioni del contratto *Vertex*.

Tutte le precedenti implementazioni sono nascoste ai client del contratto *Vertex*, utilizzando *VertexFactory* come unico mezzo per poterle costruire, esponendo un metodo statico per ogni implementazione che abbiamo descritto.

5.2.4 Pacchetto *Piping*

Questo pacchetto contiene le implementazioni che permettono di assemblare una *pipeline*, attraverso la composizione di un numero quanto si voglia di filtri.

Funzionalità implementate

Le funzionalità fornite da questo pacchetto sono le seguenti:

- definire un *template* di filtro, componente atomica per la composizione di una *pipeline*. Questo template codifica il comportamento necessario affinché possa essere combinato, lasciando all'implementatore il compito di sviluppare la logica che caratterizza il filtro che stà modellando;
- fornire un insieme di filtri già implementati necessari alla realizzazione delle funzionalità descritte nella Sezione 3.1;
- catturare ed esporre il concetto di *listener*, mediante il quale è possibile gestire degli eventi che avvengono durante la computazione. È possibile assemblare la propria pipeline ed, attraverso un listener, eseguire non solo le trasformazioni che questa produce, ma anche della logica aggiuntiva all'accadere di determinati eventi. Questo strumento permette di essere ortogonali alla pipeline ed aver maggior controllo sull'intera computazione.

Classi

Procediamo con ordine nel descrivere le idee principali catturate dalle seguenti classi:

PIPEFILTER Abbiamo cercato di formalizzare in modo preciso la più piccola unità atomica capace di eseguire una computazione all'interno di una sequenza più grande, portandoci a definire la classe *PipeFilter*: introducendola possiamo rendere il codice flessibile e trasparente, nonché più facile da implementare.

Questa classe, come abbiamo detto nei paragrafi precedenti, in realtà rappresenta solo un *template* del concetto di filtro e non esegue una trasformazione definita sull'istanza di input. Usandola è possibile assemblare una sequenza di filtri, ed eseguire la computazione propagando la richiesta di esecuzione a tutti i filtri che si sono assemblati.

Quello appena detto è stato implementato dando a *PipeFilter* una struttura ricorsiva: ad esempio per costruire una pipeline con due filtri è sufficiente costruirli, fissare la relazione di precedenza che identifica la pipeline ed invocare la richiesta di esecuzione sull'ultimo filtro. Tale filtro, ricevendo la richiesta, controlla se il filtro che lo precede esiste: se sì, delega la richiesta di esecuzione e si riapplica questo procedimento in modo ricorsivo, altrimenti esegue la trasformazione per cui è stato creato. Inoltre, sempre sfruttando la struttura ricorsiva di *PipeFilter*, è possibile comporre una pipeline i cui filtri sono a loro volta pipeline.

Ogni filtro lavora su un grafo in input, apporta le modifiche che implementa e restituisce in output il grafo trasformato. Questa trasformazione da grafi a grafi permette di non avere complessità aggiuntiva per quanto riguarda la coerenza dei formati di input/output tra filtri adiacenti.

IMPLEMENTAZIONI DI PIPEFILTER Come abbiamo detto, *PipeFilter* rappresenta solo un *template*: il comportamento che vogliamo dare ad ogni filtro deve essere codificato in una classe concreta che completa tale template. In realtà non è molto il lavoro che si lascia da scrivere: per completare il template è sufficiente implementare un metodo astratto, che ha come parametri il grafo di input e deve ritornare un grafo utilizzato come input per il filtro successivo.

Il primo filtro concreto che abbiamo introdotto è stato *PrinterPipeFilter*, il quale produce una rappresentazione grafica compilando un documento *dot*: per adempiere a questo compito, costruisce un oggetto di tipo *DotExporter* nel quale, durante la visita del grafo, colleziona le informazioni necessarie per costruire il documento da compilare. Questo filtro non apporta nessuna trasformazione al grafo di input e lo restituisce così come lo ha ricevuto.

Il secondo filtro concreto che abbiamo introdotto è stato *DfsPipeFilter*, il quale esegue una visita in profondità del grafo in input e ritorna un grafo il cui insieme di archi è composto da tutti gli archi percorsi dalla visita. Concatenando un

filtro di tipo *PrinterPipeFilter* sarà possibile visualizzare la foresta di alberi *DFS* rappresentante la visita eseguita.

Il terzo filtro concreto che abbiamo introdotto è stato *TarjanPipeFilter*, il quale ricerca le componenti fortemente connesse sul grafo di input e restituisce un nuovo *DAG* composto dalle componenti identificate. È importante notare che programmando “per interfacce”, il grafo prodotto da questo filtro non ha come vertici dei *SimpleVertex*, bensì degli oggetti che caratterizzano le componenti fortemente connesse. In questo modo possiamo continuare ad inviare gli stessi messaggi che possiamo inviare ad un oggetto di tipo *SimpleVertex*, ma ottenere un comportamento diverso, senza modificare il codice che implementa il modello di dominio.

Il quarto filtro concreto che abbiamo introdotto è stato *SourcesCollapserPipeFilter*, il quale permette di collassare tutte le sorgenti ed introdurne una nuova ed unica. Il nuovo grafo viene restituito per eventuali computazioni successive.

Gli ultimi filtri concreti che abbiamo introdotto sono stati *PlainTextStatsPipeFilter* e *ConnectedComponentInfoPipeFilter*. Questi hanno la responsabilità di rappresentare in formato testuale il grafo in input, producendo rispettivamente un output tabellare ed una struttura dati che mantiene delle informazioni sulle componenti fortemente connesse, consultabili utilizzando il relativo programma di visualizzazione che abbiamo sviluppato.

PIPEFILTERCOMPUTATIONLISTENER Il concetto di listener è molto simile a quello di *Observer* come descritto in [1], anche se nella nostra implementazione non notificiamo un cambiamento, bensì l'accadere di un determinato evento. Modelliamo un evento con dei messaggi nell'interfaccia *PipeFilterComputationListener*.

L'utilità dei listener risiede nella possibilità di ricevere degli argomenti inviati dal mittente della notifica per elaborarli nel modo desiderato. Questi argomenti possono essere relativi non solo ad uno specifico filtro, bensì ad un insieme di filtri, in quanto si usa lo stesso listener per tutta la durata della pipeline: un esempio è *PlainTextInfoComputationListener*, il quale usa una mappa, avente come chiavi oggetti di tipo filtro, e come valori informazioni sulla composizione del grafo, per distinguere gli argomenti ricevuti. Queste informazioni vengono raccolte dapprima durante l'esecuzione di un filtro *PlainTextStatsPipeFilter* e, quando il processo termina, si invia un messaggio al listener allegando le informazioni collezionate.

5.2.5 Pacchetto *Tarjan*

Questo pacchetto non contiene molte astrazioni limitandosi a definire un contratto fondamentale per implementare la visita *DFS* e l'algoritmo per la ricerca di componenti fortemente connesse.

Funzionalità implementate

Le funzionalità fornite da questo pacchetto sono le seguenti:

- definire il contratto che stabilisce gli eventi salienti che avvengono durante la visita in profondità del grafo. Questi eventi sono codificati come messaggi che possono essere inviati ad un implementatore del contratto, notificando lo stato in cui si trova la visita. Questo permette di introdurre i punti di estensione come descritto nella Sezione 2.2.4;
- fornire due implementatori del contratto descritto nel punto precedente: uno per costruire un albero *DFS*, l'altro per costruire il meta grafo ottenuto ricercando le componenti fortemente connesse.

Classi

Procediamo con ordine nel descrivere le idee principali catturate dalle seguenti classi:

DFSEVENTSLISTENER Questo contratto definisce gli stati, in cui la visita in profondità può entrare, necessari alle nostre implementazioni. Riportiamo sotto la sua definizione direttamente dal relativo file sorgente:

```
package tarjan;
public interface DfsEventsListener {

    void searchCompleted (Map<Vertex ,
        ExploreStatedWrapperVertex> map);
    void postVisit (Vertex v);
    void preVisit (Vertex v);
    void searchStarted (Map<Vertex ,
        ExploreStatedWrapperVertex> map);
    void newVertexExplored (Vertex explorationCauseVertex ,
        Vertex
9      vertex);
    void fillCollectedVertices (Set<Vertex> vertices);
    void alreadyKnownVertex (Vertex vertex);
}
```

Il motore che utilizza quest'interfaccia è la classe *OurModel*. Con la definizione di questo contratto possiamo implementare la visita in profondità in stile orientato agli oggetti, allontanandosi da una più comune implementazione procedurale. Crediamo sia interessante vederla, per cui ci dilungheremo in questi paragrafi nella sua descrizione. L'inizio della computazione è il seguente metodo definito nella classe *OurModel*:


```

public OurModel runDepthFirstSearch(DfsEventsListener
dfsEventsListener) {
3   final Map<Vertex, ExploreStatedWrapperVertex> map =
      makeDfsVertexMetadataMap();
      dfsEventsListener.searchStarted(map);
      for (Entry<Vertex, ExploreStatedWrapperVertex> entry :
        map.entrySet()) {
          entry.getValue().ifNotExplored(dfsEventsListener,
            new
8          ExploreStateWrapperVertexMapper()
            {
              @Override
              public ExploreStatedWrapperVertex map(Vertex vertex) {
                return map.get(vertex);
              }
13          });
        }
      dfsEventsListener.searchCompleted(map);
      return this;
    }

```

Come si nota, mancano alcuni pezzi per una visita corretta, incapsulati nella classe *ExploreStateWrapperVertex*. Abbiamo preso questa decisione per avere un sistema più modulare, rispetto a codificare tutto in un unico metodo. La responsabilità di quest'ultima classe è di associare ad ogni vertice l'informazione se questo è stato visitato oppure no durante la visita, in modo da "chiedere" ai vertici non ancora visitati di esplorare il loro vicinato, inviando il messaggio *ifNotExplored* che riportiamo:

```

public class ExploredStateWrapperVertex...
public ExploreStatedWrapperVertex ifNotExplored(
3   final DfsEventsListener dfsEventsListener,
      Vertex explorationCauseVertex,
      final ExploreStateWrapperVertexMapper mapper) {
      final Vertex vertex = getWrappedVertex();
      if (isExplored() == false) {
8         toggle();
         if (explorationCauseVertex != null) {
             dfsEventsListener.newVertexExplored(
                 explorationCauseVertex, vertex);
         }
         dfsEventsListener.preVisit(vertex);
    }

```

```

13      vertex.doOnNeighbors(new
        VertexLogicApplierWithNeighborhoodRelation() {
            @Override
            public void apply(Vertex parent, Vertex neighbour)
                {
                    if ((parent == vertex) == false) {
18                        throw new RuntimeException("Semantic error");
                    }
                    mapper.map(neighbour).ifNotExplored(
                        dfsEventsListener, parent, mapper);
                }
            });
            dfsEventsListener.postVisit(vertex);
23    } else {
        dfsEventsListener.alreadyKnownVertex(vertex);
    }
    return this;
}

```

Questo è il blocco mancante nel metodo precedente e che effettivamente caratterizza la visita in profondità. Nel metodo sopra riportato si nota che è stato semplice implementare questa parte in quanto, data la natura ricorsiva della struttura, non dobbiamo far altro che invocare lo stesso metodo su tutti i vertici del vicinato. Sarà in base al loro stato che l'invocazione si propagherà ai rispettivi vicinati.

Facciamo un'ultima osservazione riguardo agli eventi notificati: i listener concreti non sono obbligati a definire della logica per ogni evento, in quanto ognuno di questi viene creato per implementare una specifica variante e non necessariamente ogni evento è richiesto per raggiungere quanto desiderato.

DFSEVENTSLISTENERTREEBUILDER Questo listener associa ad ogni vertice delle informazioni necessarie per la costruzione dell'albero *DFS*. In particolare si mantiene una coppia di istanti (t_{in} , t_{out}) dove t_{in} rappresenta l'istante in cui il vertice viene raggiunto e t_{out} l'istante in cui si è finito di visitare il rispettivo vicinato.

Inoltre, nella gestione dell'evento *newVertexExplored*, si costruisce la nuova relazione di vicinato, includendo solo quegli archi che hanno raggiunto vertici non ancora visitati.

Riprendendo l'osservazione fatta al termine del paragrafo precedente, questo listener non ha bisogno di definire nessuna logica per l'evento *alreadyKnownVertex*.

TARJANEVENTSLISTENER**TREEBUILDER** Questo listener implementa l'algoritmo per la ricerca delle componenti fortemente connesse, utilizzando la variante descritta nella Sezione 2.3.3.

Questo grafo contiene come vertici degli oggetti che hanno comportamento specifico relativo alle componenti fortemente connesse per cui, nelle successive computazioni, sarà possibile usare il grafo in modo trasparente, inviando messaggi polimorfi che produrranno il comportamento desiderato (ad esempio la l'etichetta nella rappresentazione grafica sarà diversa rispetto a quella che si può ottenere dopo una visita in profondità).

BIBLIOGRAFIA

- [1] Alpert, Brown, Woolf, *The Design Pattern Smalltalk Companion*. Addison Wesley, Massachusetts, 1st Edition, 1998.
- [2] Kent Beck, *Test-Driven Development: by Example*. Addison Wesley, Massachusetts, 1st Edition, 2003
- [3] Various Contributors, *SBML Official Documents*.
<http://sbml.org/Documents>
- [4] Various Contributors, *JSBML open source java library*.
<http://sbml.org/Software/JSBML>
- [5] Crescenzi, Marino et al., *Telling Stories*.
- [6] Borenstein, Kupiec, Feldman, Ruppín, *Large-scale reconstruction and phylogenetic analysis of metabolic environments*.
- [7] Dasgupta, Papadimitriou, Vazirani, *Algorithms*. McGraw-Hill, 1st Edition, 2008.
- [8] Crescenzi, Gambosi, Grossi, *Strutture di dati e algoritmi*. Pearson Education Addison-Wesley, 2006, ISBN 8871922735.
- [9] Buschmann, Meunier, Rohnert, Sommerlad, Stal, *Pattern-Oriented Software Architecture*. John Wiley and Sons Ltd, 1996.
- [10] Coplien, Schmidt, *Program Language Of Program Design*. Addison Wesley, 1995.
- [11] Tarjan, *Depth-First Search and linear graph algorithms*
- [12] Various Contributors, <http://pbil.univ-lyon1.fr/software/symbiocyc/index.php>
- [13] Various Contributors, <http://metexplore.toulouse.inra.fr/metexploreJoomla/>
- [14] Massimo Nocentini, <https://github.com/massimo-nocentini/my-undergraduatethesis-j>