Università di Pisa

Dipartimento di Informatica

Corso di Laurea Triennale

SBML-batch: un package Python per la simulazione in batteria di reti di reazioni chimiche

Candidata:
Mariagiovanna Rotundo

Relatori:
Prof. Paolo Milazzo
Dott.ssa Lucia Nasti

Anno Accademico 2019/2020

Indice

Capitolo 1

Introduzione

Il tirocinio descritto in questa tesi è stato svolto presso il Dipartimento di Informatica dell'Università di Pisa. È stato iniziato a febbraio e terminato a giugno 2020. L'obiettivo è quello di creare un package Python, pronto per l'installazione e l'utilizzo da parte di terzi, che si occupi dello studio della struttura e la successiva simulazione in batteria di modelli biologici e biochimici descritti tramite il linguaggio SBML. La gestione della simulazione varia in base alle informazioni ricavate dall'analisi precedente.

La creazione di questo package ha lo scopo di permettere uno studio del modello utilizzando un approccio *in silico* [20].

Esistono tre tipi di approcci allo studio di sistemi biologici: *in vivo*, *in vitro* e *in silico*. Nei primi due, i fenomeni considerati vengono studiati con un'osservazione diretta. In particolare, il primo osserva il fenomeno mentre si verifica in un organismo vivente, mentre il secondo lo osserva attraverso colture cellulari fatte in laboratorio, in provetta [23]. Il terzo approccio analizza, invece, i fenomeni biochimici attraverso simulazioni che vengono eseguite al computer [23]. Per fare ciò le proprietà e le funzionalità delle reti biologiche vengono astratte attraverso l'implementazione di algoritmi e di modelli computazionali. Sono, quindi, stati sviluppati formalismi per rappresentare tali sistemi. Un modo per esaminare una rete di reazioni è pensarla come un grafo e analizzarne la topologia. Una funzione implementata nel progetto, che utilizza questa rappresentazione, studia il grafo per cercare di ottenere l'andamento del modello applicando dei teoremi esistenti in letteratura, in particolare i teoremi formulati da Feinberg. Un altro formalismo, utilizzabile per lo studio *in silico*, è quello delle Petri nets [25] che rappresenta un insieme di reazioni chimiche come un grafo che presenta due tipi di nodi distinti, che rappresentano molecole e reazioni, i quali sono connessi tra loro mediante archi orientati. Nel software viene usata un'estensione delle Petri nets che presenta diversi tipi di archi così da poter rappresentare i diversi ruoli delle specie all'interno delle reazioni presenti nella rete.

L'approccio *in silico* permette, inoltre, di processare in tempi ragionevoli grandi quantità di dati ottenuti mediante lo studio dei fenomeni biochimici *in vitro* e *in vivo*. In alcuni casi ciò ha permesso di ottenere informazioni circa

i sistemi studiati evidenziandone caratteristiche che non erano state notate in precedenza.

Da tutto ciò si evince che l'informatica può aiutare la ricerca in ambito biologico in modi differenti. Nel package implementato vengono analizzati e simulati *in silico* modelli le cui informazioni derivano da studi ed esperimenti precedenti. Un esempio di queste informazioni sono i valori delle constanti cinetiche associate alle reazioni, già definite al momento della simulazione della rete.

Il software creato permette di studiare la topologia di reti di reazioni chimiche e di simularle. Lo studio della struttura della rete ha lo scopo di acquisire maggiori informazioni, riguardanti le sue proprietà dinamiche, il suo comportamento, le quali saranno usate per guidare le simulazioni. Per fare ciò sono stati utilizzati due teoremi presenti in letteratura formulati da Feinberg: il *Deficiency Zero Theorem* e il *Deficiency One Theorem* [8], i quali ci assicurano, per modelli con determinate caratteristiche, il raggiungimento dello stato stazionario.

Nei casi in cui questi teoremi non ci possano dare informazioni riguardo un sistema, è la simulazione stessa che cerca di capirne l'andamento, cercando di individuare oscillazioni o il raggiungimento dello steady state. Per ogni rete di reazioni chimiche vengono eseguite più simulazioni considerando perturbazioni alle quantità iniziali. Tutti i dati generati dalle simulazioni vengono memorizzati.

Dopo aver studiato e simulato i modelli biologici e biochimici, ognuno di essi viene formalizzato mediante la corrispondente Petri net, la quale rappresenta la rete come un grafo bipartito.

Lo scopo della simulazione in batteria di modelli biologici, memorizzandone i risultati delle simulazioni e le Petri nets, è quello di creare un dataset composto da queste informazioni. Questo in futuro potrà essere usato per studiare le proprietà dinamiche delle reti e correlarle alle Petri nets. Ad esempio, potrà essere usato per studiare la robustezza dei sistemi. Un sistema è robusto se, nonostante la presenza di perturbazioni iniziali, il suo comportamento rimane invariato. Un modo in cui potrà essere studiata questa proprietà, potrebbe consistere nell'usare il dataset per allenare un modello di Machine Learning [5], il quale sarà in grado di generalizzare (con un certo grado di accuratezza) su dati non visti, ovvero, sarà capace di dire se un sistema, non presente nel dataset usato per il training, presenta la proprietà di robustezza [5].

La topologia della rete viene analizzata attraverso lo studio della sua Petri net, la quale non prevede la rappresentazione di informazioni non mostrate nella struttura della rete stessa. Ad esempio non permette di rappresentare eventi o regole contenute nel sistema. Per questo motivo verranno selezionati per l'analisi solo modelli che non presentano né regole, né eventi. Inoltre non verranno considerati nemmeno reti qualitative, un'estensione delle reti booleane dove i processi non vengono descritti come reazioni chimiche [9].

Il seguito di questa relazione è organizzato come segue.

Nel Capitolo 2 vengono illustrati i concetti di base relativi alle reazioni chimiche e le formalizzazioni usate all'interno del progetto. Inoltre, vengono enunciati

i teoremi di Feinberg, sopra citati, utilizzati per lo studio della topologia di reti di reazioni chimiche così da ottenere maggiori informazioni su di esse.

Nel Capitolo 3 vengono illustrati gli strumenti software e i linguaggi utilizzati. Tra questi ultimi compaiono l'SBML e il linguaggio DOT, utilizzati per la rappresentazione di sistemi biologici e delle Petri nets ad essi associate.

Nel Capitolo 4 viene descritta in modo dettagliato l'implementazione del software, illustrando ogni script di cui è composto, e le caratteristiche del package creato.

Nel Capitolo 5 vengono illustrati i parametri utilizzati all'interno del progetto e le analisi che sono state effettuate per la loro scelta.

Nel Capitolo 6 vengono mostrate le conclusioni del lavoro svolto.

Capitolo 2

Background

In questo capitolo verranno enunciati i concetti e gli approcci che sono alla base del progetto realizzato. Nella sezione 2.1 verranno descritti i concetti di base inerenti alle reazioni chimiche; nella sezione 2.2 verranno illustrati i possibili approcci per studiare le reti di reazioni chimiche; nella sezione 2.3 verranno descritti i teoremi di Feinberg che permettono di studiare l'andamento di un sistema biologico analizzandone la struttura; nella sezione 2.4 verrà introdotto il formalismo delle Petri nets, utilizzato per rappresentare una rete di reazioni chimiche come un grafo bipartito.

2.1 Le reazioni chimiche

Una reazione chimica, fisicamente, è uno scontro tra particelle di una o più specie chimiche le quali subiscono una trasformazione.

Possiamo rappresentare una reazione chimica come un'equazione:

$$A + 2B \xrightarrow{k_1} C + D \tag{2.1}$$

Dove A e B sono le specie chimiche che innescano la reazione, note come *reagenti*, mentre C e D sono le specie chimiche prodotte dalla reazione, note come *prodotti*. Se una specie compare in una reazione come reagente è possibile che appaia anche come prodotto, in quanto potrebbe attivare una reazione senza venire consumata da essa, come, ad esempio, la specie A nella reazione 2.2:

$$A + 2B \xrightarrow{k_1} A + C \tag{2.2}$$

La freccia rappresenta il verso della reazione, cioè permette di individuare quali sono i reagenti e quali invece i prodotti.

Ad ogni specie è associato un *coefficiente stochiometrico*, un valore che ne indica la molteplicità, cioè il numero di molecole con cui la specie partecipa alla reazione. Nella reazione 2.1, il coefficiente stochiometrico associato ad A, C e D è 1, mentre quello associato a B è 2.

Infine, il parametro k_1 rappresenta la costante cinetica associata alla reazione, un valore che influisce sulla sua velocità.

In un'equazione chimica può essere presente una sola reazione (come nell'equazione 2.1) oppure possono essercene due, una inversa all'altra, che rappresentano processi di trasformazione opposti. Nel primo caso la reazione si dice *irreversibile*, mentre nel secondo *reversibile*.

Un esempio di reazione reversibile è il seguente:

$$A + 2B \underset{\mathrm{k_{-1}}}{\overset{\mathrm{k_1}}{\rightleftharpoons}} C + D \tag{2.3}$$

in cui il parametro k_{-1} rappresenta la costante cinetica associata alla reazione inversa.

Un insieme di reazioni chimiche è detto Chemical Reaction Network (CRN).

2.2 Modelli deterministici e stocastici

Un sistema di reazioni chimiche può essere analizzato attraverso la realizzazione di un modello matematico, con lo scopo di descrivere e comprendere il funzionamento del sistema a livello molecolare. Dal punto di vista computazionale, il sistema può essere studiato applicando principalmente due approcci [4]:

- l'approccio deterministico
- l'approccio stocastico

L'approccio deterministico si è dimostrato molto efficace in chimica e in biochimica. Alla base c'è la legge *di azione di massa*, una legge empirica che fornisce una relazione tra velocità e concentrazione delle specie, affermando che la velocità di una reazione è proporzionale ai prodotti delle concentrazioni delle molecole partecipanti. La velocità di una reazione, studiata mediante questa legge, è uguale al prodotto della costante cinetica e delle concentrazioni delle specie reagenti che presentano un esponente pari al loro coefficiente stochiometrico [8]. Consideriamo, come esempio, la reazione 2.1. La sua velocità viene calcolata nel modo seguente [14]:

$$k_1[A]^1[B]^2 \tag{2.4}$$

dove [A], convenzionalmente, rappresenta la concentrazione della specie A e [B] la concentrazione della specie B. Se la reazione è reversibile, la velocità dell'inversa viene calcolata in modo analogo.

Le velocità delle reazioni possono essere usate per definire delle equazioni differenziali (ODE, Ordinary Differential Equation) una per ogni specie presente nella rete. Ogni equazione descrive come varia nel tempo la concentrazione della specie rappresentata.

Questo approccio permette di analizzare l'evoluzione temporale del modello da un punto di vista macroscopico. Ciò vuol dire che stiamo assumendo che [4]:

1. il sistema è omogeneo;

2. il numero di molecole di ogni specie coinvolta è "sufficientemente" grande;

Questo ci permette di studiare il sistema approssimando il suo comportamento e smorzando la casualità delle interazioni molecolari.

A differenza dell'approccio deterministico, l'approccio stocastico si basa sull'idea che le reazioni sono processi casuali e discreti dovuti a collisioni molecolari, che hanno una certa probabilità di verificarsi. Un algoritmo usato per simulare un sistema utilizzando un approccio stocastico è l'algoritmo definito da Gillespie [22], il quale calcola una possibile evoluzione del sistema. Risulta particolarmente efficiente per la simulazione di reazioni che presentano un basso numero di reagenti.

Questi due approcci analizzano un sistema biologico da punti di vista differenti. Nella Figura 2.1, si può osservare l'uso dei due approcci su uno stesso modello, supponendo le specie, le quantità iniziali e le reazioni elencate nella Tabella 2.1.

Specie	Quantità iniziali
Glu	160
Fru	0
Formic_acid	0
Triose	0
Acetic_acid	0
Cn	0
Amadori	0
AMP	0
C5	0
lys_R	15
Melanoidin	0

(a) Specie e quantità iniziali

Reazioni
$\text{Glu} \xrightarrow{K_1} \text{Fru}$
$\text{Fru} \xrightarrow{K_2} \text{Glu}$
$\text{Glu} \xrightarrow{K_3} \text{C5+Formic\_acid}$
$\text{Fru} \xrightarrow{K_4} \text{C5+Formic\_acid}$
$\text{Fru} \xrightarrow{K_5} \text{2Triose}$
$\text{Triose} \xrightarrow{K_6} \text{Cn + Acetic\_acid}$
$\text{lys\_R+Glu} \xrightarrow{K_7} \text{Amadori}$
$\text{Amadori} \xrightarrow{K_8} \text{Acetic\_acid+lys\_R}$
$\text{Amadori} \xrightarrow{K_9} \text{AMP}$
$\text{lys\_R+Fru} \xrightarrow{K_{10}} \text{AMP}$
$\text{AMP} \xrightarrow{K_{11}} \text{Melanoidin}$

(b) Reazioni

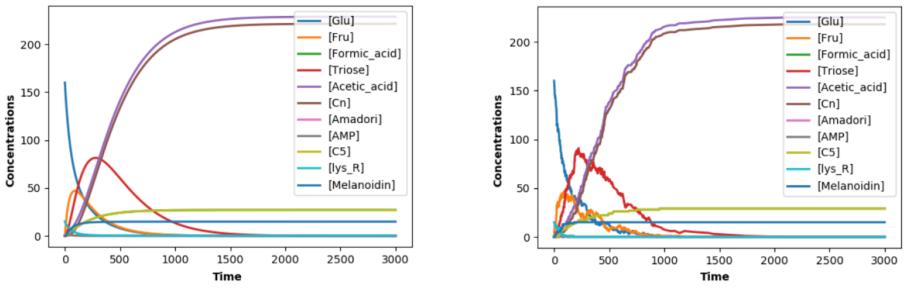
Tabella 2.1: Caratteristiche del modello.

Lo stato stazionario

Quando la velocità di una reazione è uguale alla velocità dell'inversa la reazione è detta essere in equilibrio dinamico. Quando ciò accade, le quantità delle specie che compaiono nelle reazioni smettono di variare e si può avere l'impressione che nel sistema non si stia più verificando alcuna reazione. In realtà le reazioni continuano ad avere luogo, ma si compensano l'una con l'altra.

Una rete di reazioni può raggiungere una condizione detta di *steady state* quando la quantità di ogni specie coinvolta nella rete rimane costante.

Figura 2.1: La figura a sinistra mostra il comportamento di un modello utilizzando un approccio deterministico, mentre la figura a destra mostra lo stesso modello analizzato con un approccio stocastico.



In generale, la maggior parte dei modelli biologici raggiunge lo steady state. Uno stesso modello, con velocità e condizioni iniziali diverse, potrebbe raggiungere stati stazionari differenti oppure raggiungere lo stesso stato stazionario in un tempo differente.

Nell'evoluzione di un sistema che raggiunge lo steady state possiamo individuare due periodi: il *transient period* che identifica il tempo trascorso dallo stato iniziale fino al raggiungimento dello stato stazionario, lo stato stazionario che rappresenta il tempo in cui il sistema ha raggiunto la fase di equilibrio [15].

La Figura 2.2 rappresenta l'evoluzione di un sistema che presenta le specie, le quantità iniziali e le reazioni elencate nella Tabella 2.1.

Il comportamento di un sistema biologico può essere analizzato eseguendo delle simulazioni. Simulare una rete di reazioni è costoso in termini computazionali e può richiedere molto tempo, in quanto il modello andrebbe simulato e studiato considerando le diverse configurazioni iniziali.

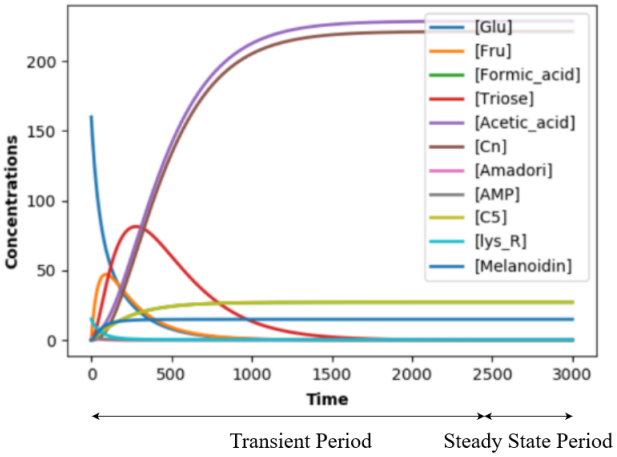
Sono stati svolti numerosi studi per poter analizzare il comportamento di un sistema senza dover ricorrere alla sua simulazione e, a questo proposito, sono presenti in letteratura due teoremi formulati da Feinberg: il *Deficiency Zero Theorem* e il *Deficiency One Theorem* [8].

2.3 I teoremi di Feinberg

Il *Deficiency Zero Theorem* e il *Deficiency One Theorem* [8] ci permettono di dire se un sistema raggiunge lo steady state, indipendentemente dalla configurazione iniziale considerata, senza ricorrere alla sua simulazione, ma analizzandone la struttura. Useremo, quindi, questi teoremi per cercare di capire se una rete raggiunge lo stato stazionario.

Introduciamo, innanzitutto, alcune definizioni che verranno usate nei due teoremi.

Figura 2.2: Transient period e stato stazionario di un modello biologico.



Per studiare le proprietà strutturali di una Chemical Reaction Network, possiamo formalizzarla come un grafo orientato. Formalmente, un grafo orientato è una coppia ordinata (V,E) dove V è un insieme finito di vertici ed E è un insieme finito di archi che esprimono relazioni tra coppie di nodi [19]. Nel grafo rappresentante una CRN i nodi rappresentano i gruppi composti dai reagenti o dai prodotti, gli archi rappresentano le reazioni e sono orientati nello stesso verso di queste ultime. Gruppi in cui appaiono esattamente le stesse specie, ma con coefficienti stochiometrici diversi vengono considerati come nodi differenti del grafo. L'ordine delle specie non è, invece, rilevante.

Nel caso in cui una specie venga prodotta oppure decada in modo spontaneo, viene aggiunta una specie fittizia "0" come reagente, nel primo caso, oppure come prodotto, nel secondo. Ad esempio, se una certa specie A viene prodotta in modo spontaneo, ciò viene descritto attraverso la reazione

$$0 \xrightarrow{k} A \tag{2.5}$$

Consideriamo, invece, come esempio la seguente rete di reazioni [15]:

$$\begin{aligned} A + B &\xrightarrow{k_1} 2B \\ B &\xrightarrow{k_2} A \end{aligned} \tag{2.6}$$

In Figura 2.3 è rappresentato il grafo costruito considerando il sistema di reazioni 2.6. Qui è possibile identificare:

- i nodi del grafo A+B, 2B, B, A;
- le *linkage classes* che rappresentano le componenti connesse del grafo. In questo caso {A+B, 2B} e {B, A}.

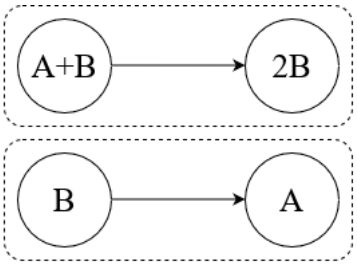


Figura 2.3: Grafo rappresentante una rete di reazioni.

Un grafo può essere descritto formalmente da strutture dati differenti. Tra queste compaiono la *matrice di incidenza* e le *liste di adiacenza*. Una matrice d'incidenza [19] è una matrice M di dimensione

$$\#V * \#E \tag{2.7}$$

tale che

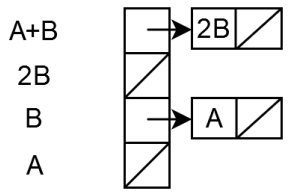
$$M_{ij}=\begin{cases} \text{-1 se l'arco } j \text{ esce dal vertice } i; \\ \text{1 se l'arco } j \text{ entra nel vertice } i; \\ \text{0 negli altri casi.} \end{cases}$$

La matrice d'incidenza corrispondente al grafo in Figura 2.3 sarà così composta:

$$\begin{array}{c} \\ \text{A+B} \\ \text{2B} \\ \text{B} \\ \text{A} \end{array} \begin{array}{c} \begin{array}{cc} R_1 & R_2 \end{array} \\ \begin{pmatrix} \text{-1} & 0 \\ 1 & 0 \\ 0 & \text{-1} \\ 0 & 1 \end{pmatrix} \end{array}$$

Una lista di adiacenza, associata a un nodo i, è una lista che contiene i vertici v per cui esiste l'arco *(i,v)*. Un grafo rappresentato con liste di adiacenza, presenta una lista per ogni vertice, quindi queste sono in numero pari alla #V [19].

Considerando ancora il grafo in Figura 2.3, questo sarà definito dalle seguenti liste di adiacenza:



Feinberg [8] descrive ed analizza le reti di reazioni utilizzando concetti e teoremi appartenenti all'algebra lineare e rappresenta i nodi del grafo corrispondente a una CRN con dei vettori, uno per ogni nodo [8]. Questi vettori sono costruiti associando agli elementi del vettore i coefficienti stochiometrici delle specie che appaiono nel nodo.

Considerando il grafo in Figura 2.3 e supponendo che le specie siano ordinate in ordine alfabetico, i vettori associati ai nodi sono A+B=[1,1], 2B=[0,2], B=[0,1], A=[1,0].

È possibile usare questa rappresentazione dei nodi per costruire un altro tipo di matrice associata a una CRN, la *matrice stochiometrica*, in cui ogni riga rappresenta una reazione e ogni colonna rappresenta una specie [8].

Al grafo in Figura 2.3 corrisponde una matrice stochiometrica di dimensione 2*2, in quanto nella rete sono presenti 2 reazioni e 2 specie.

I vettori "reazione" sono ottenuti sottraendo i vettori, che rappresentano i reagenti, ai vettori che rappresentano i prodotti della reazione.

Otteniamo quindi i seguenti vettori:

$$\begin{gathered} 2B - (A + B) = [0, 2] - [1, 1] = [-1, 1] \\ A - B = [1, 0] - [0, 1] = [1, -1] \end{gathered} \tag{2.8}$$

Dai vettori reazione, otteniamo la seguente matrice stochiometrica:

$$\begin{array}{c} \\ R_1 \\ R_2 \end{array} \begin{array}{c} \begin{array}{cc} \text{A} & \text{B} \end{array} \\ \begin{pmatrix} \text{-1} & 1 \\ 1 & \text{-1} \end{pmatrix} \end{array}$$

A questo punto è possibile calcolare la *deficiency* della rete, un numero intero non negativo che rappresenta il grado di indipendenza lineare delle reazioni della rete. È definito nel modo seguente:

$$deficiency = n - l - rank(Ms) \tag{2.9}$$

dove n è il numero di nodi, l il numero di linkage classes e *rank(Ms)* è il rango della matrice stochiometrica.

Analizzando la rete 2.6, si può osservare che la deficiency è 1 in quanto:

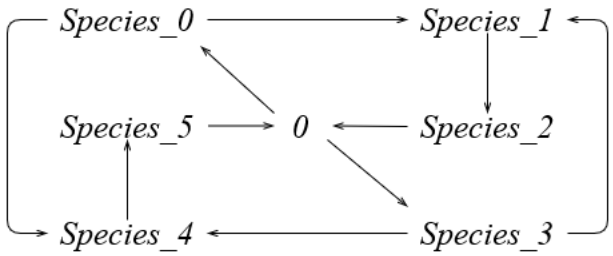
$$deficiency = n - l - rank(Ms) = 4 - 2 - 1 = 1 \tag{2.10}$$

Feinberg nell'articolo [8], infine, introduce i concetti di rete *reversible* e rete *weakly reversible*.

Una rete di reazioni si dice *reversible* se per ogni coppia di nodi del grafo orientato *x, y* tali che esiste l'arco *(x,y)* allora esiste l'arco *(y,x)*, o, in altre parole, se per ogni reazione esiste la sua inversa.

Una rete di reazioni si dice, invece, *weakly reversible* se per ogni coppia di nodi del grafo *x, y* tali che esiste l'arco *(x,y)* allora esiste un percorso sul grafo, che segue la direzione degli archi, tale per cui x è raggiungibile da y.

Figura 2.4: Una rete weakly reversible.



Si può osservare che la rete di reazioni 2.6 non è né *reversible* (in quanto non sono presenti reazioni inverse) né *weakly reversible* in quanto da A+B possiamo raggiungere 2B e da B possiamo raggiungere A, ma il viceversa non è possibile.

La Figura 2.4 rappresenta una rete *weakly reversible* composta dalle specie e dalle reazioni elencate nella Tabella 2.2.

Specie	Reazioni
species_0	species_0 → species_1
species_1	0 → species_0
species_2	species_1 → species_2
species_3	0 → species_3
species_4	species_3 → species_4
species_5	species_4 → species_5
0	species_0 → species_4
	species_3 → species_1
	species_2 → 0
	species_5 → 0

Tabella 2.2: Specie e reazioni di una CRN.

I concetti sopra descritti verranno usati nei due teoremi formulati da Feinberg: il *Deficiency Zero Theorem* e il *Deficiency One Theorem*, che ci permettono di analizzare la struttura di una CRN così da avere maggiori informazioni riguardo il suo comportamento [8]. In particolare ci dicono che se una rete di reazioni, descritta mediante la legge di azione di massa, è reversible o weakly reversible e presenta una deficiency pari a 0 oppure pari a 1 allora raggiunge lo steady state, indipendentemente dal valore posseduto dalle costanti cinetiche [8]. Il *Deficiency Zero Theorem* ci garantisce che lo stato stazionario è unico, mentre il *Deficiency One Theorem* ci assicura la sua esistenza, ma non l'unicità.

2.4 Le Petri nets

In letteratura, esistono molti formalismi matematici che possono essere utilizzati per descrivere e analizzare i sistemi biologici, evidenziandone le caratteristiche peculiari. Tra questi compare il formalismo delle *Petri nets* che ci consente di modellare le reti di reazioni chimiche e visualizzare in forma grafica la topologia della rete e le sue proprietà strutturali e di dimostrarne la correttezza attraverso concetti della teoria dei grafi. Questa rappresentazione ci permette di mostrare in modo esplicito gli eventi del sistema che possono essere indipendenti l'uno dall'altro [25].

Formalmente, una Petri net è un grafo bipartito orientato che presenta due tipologie di nodi: i *posti* e le *transizioni*. Gli archi del grafo connettono tra loro solo nodi di categorie diverse, cioè collegano soltanto posti a transizioni e viceversa [24].

Una Petri nets modella una CRN rappresentandone le specie e le reazioni mediante i posti, nel primo caso, e le transizioni, nel secondo.

Per descrivere al meglio una rete di reazioni viene utilizzata un'estensione che prevede altre due tipologie di archi permettendo, in questo modo, di rappresentare anche i *modificatori* delle reazioni. Una specie si comporta da modificatore per una certa reazione se ne influenza la velocità, ma non agisce né da reagente né da prodotto per essa. In questo caso si parla di *promotore* se "aiuta" la reazione, cioè se con l'aumentare della sua concentrazione la velocità aumenta, mentre si parla di *inibitore* se la velocità, invece, diminuisce.

In un sistema, una specie può comportarsi come promotore per una reazione, ma come inibitore per un'altra, in quanto il suo comportamento dipende dalla reazione considerata.

Graficamente una Petri nets rappresentante una CRN viene dunque definita secondo i seguenti criteri:

1. i posti sono rappresentati come cerchi e le transizioni come rettangoli;

2. tutti i nodi-specie sono collegati mediante un arco orientato ai nodi-reazione. Se la specie è un reagente della reazione allora viene aggiunto un arco uscente dal nodo rappresentante la specie ed entrante nel nodo che rappresenta la reazione. Se la specie è un prodotto della reazione allora viene aggiunto un arco entrante nel nodo-specie ed uscente dal nodo-reazione. Agli archi è associata un'etichetta numerica che corrisponde al coefficiente stochiometrico con il quale la specie dà luogo alla reazione oppure è prodotta da essa;

3. se una specie si comporta da promotore per una certa reazione viene aggiunto un arco tra i due nodi che presenta un cerchio pieno dal lato della reazione;

4. se una specie si comporta da inibitore per una certa reazione viene aggiunto un arco tra i due nodi che presenta una barra dal lato della reazione.

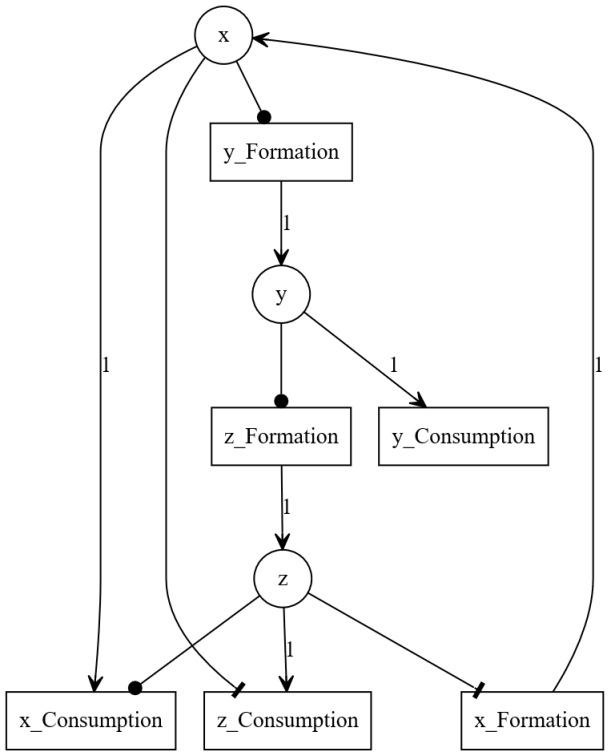
Specie	Nome della reazione	Reazione	Modificatori della reazione
x	x_Formation	$\emptyset \rightarrow$ x	z
y	x_Consumption	x $\rightarrow \emptyset$	z
z	y_Formation	$\emptyset \rightarrow$ y	x
	y_Consumption	y $\rightarrow \emptyset$	
	z_Formation	$\emptyset \rightarrow$ z	y
	z_Consumption	z $\rightarrow \emptyset$	x

Tabella 2.3: Specie e reazioni di una rete di reazioni. $\emptyset$ indica che non ci sono reagenti o prodotti.

Nella Figura 2.5 si può osservare un esempio di Petri net che modella una rete che presenta le specie e le reazioni riportate nella Tabella 2.3.

Guardando la figura e analizzando il ruolo della specie x all'interno del sistema si può osservare che essa è un reagente con coefficiente stochiometrico pari a 1 per la reazione "x_Consumpion" ed è un prodotto, ancora con molteplicità 1, per la reazione "x_Formation". Inoltre, si comporta come promotore per la reazione "y_Formation" e come inibitore per la reazione "z_Consumption".

Figura 2.5: Esempio di Petri net di un sistema biologico.



Capitolo 3

Strumenti utilizzati

Lo scopo del tirocinio è realizzare un pacchetto in linguaggio Python che permetta di analizzare e simulare il comportamento di insiemi di reazioni chimiche che presentano caratteristiche specifiche. Un package è una directory contenente uno script chiamato `__init__.py` e altri moduli Python o altri package.

I modelli analizzati sono stati scaricati da *BioModels* [11], un archivio online che contiene modelli di sistemi biologici e biomedici consultabili gratuitamente. Sono disponibili modelli di diverso tipo che si differenziano, ad esempio, per il formato usato per rappresentare i dati oppure per il tipo di approccio usato nel modello. I modelli di cui siamo interessati a studiare il comportamento sono i modelli curati manualmente e descritti utilizzando l'*SBML*.

L'SBML (Systems Biology Markup Language) è un linguaggio basato su XML (eXtensible Markup Language, un linguaggio di markup) che viene spesso usato per rappresentare fenomeni biologici e biochimici. Presenta una struttura e una sintassi descritte nella Figura 3.1 [13].

Un modello è composto da liste, le quali sono composte a loro volta da un certo numero di elementi. Gli elementi definiti in una lista possono essere usati nella definizione degli elementi di un'altra. Ad esempio, trasformando in SBML la reazione 2.1 si può ottenere un modello risultante come quello in Figura 3.2, dove sono stati usati valori fittizi.

Ogni elemento di un certo tipo viene descritto tramite determinati attributi del linguaggio SBML. Nella Tabella 3.1 si possono osservare gli attributi per definire un elemento di tipo specie e quelli per definire, invece, un elemento di tipo parametro [13].

Non tutti gli attributi sono obbligatori, ma alcuni sono opzionali. Inoltre, tra di essi possono essere presenti dei vincoli. Un esempio di attributo opzionale è l'attributo booleano "`reversible`" associato alle reazioni [13]. Questo permette di specificare se una reazione è reversibile oppure no e, nel primo caso, permette di non dover descrivere l'inversa in modo esplicito in quanto può essere ricavata dalla descrizione della reazione analizzata. In alcuni casi, anche se tale attributo non è presente, la reversibilità di una reazione può essere dedotta analizzando come viene calcolata la sua velocità. Questo può far sì che

Figura 3.1: Schema di un modello SBML come descritto in [13].

ci siano informazioni contraddittorie, ma ciò viene considerato come un errore nella codifica del modello [13].

I modelli di cui ci interessa studiare il comportamento sono modelli che non presentano regole o eventi, in quanto questi non possono essere rappresentati nelle Petri nets. Le regole permettono di definire comportamenti e relazioni tra le componenti della rete che non possono essere espresse con le reazioni. Gli eventi rappresentano avvenimenti a cui possono essere soggette le componenti del sistema che possono presentarsi anche con un certo ritardo rispetto alla condizione che li provoca [13].

Quindi, osservando le Petri nets, non è possibile notare come queste caratteristiche del sistema influenzino le quantità delle specie a tempo di simulazione. Non sono di nostro interesse nemmeno i modelli qualitativi, cioè modelli in cui sono descritti parametri logici utilizzando un'estensione dell'SBML e in cui i processi non vengono descritti come reazioni chimiche [9].

libSBML e libRoadRunner

Per poter analizzare un modello SBML e poter estrarre da questo le informazioni di nostro interesse, è stata utilizzata la libreria *libSBML* [9], una libreria gratuita e open-source.

Figura 3.2: Esempio di modello SBML.

```
<?xml version="1.0" encoding="UTF-8"?>
<sbml xmlns="http://sbml.org/Documents/Specifications" level="3" version="2">
  <model id="example" name="example_model">
    <listOfSpecies>
        <species  id="A" initialAmount="3" />
        <species  id="B" initialAmount="4" />
        <species  id="C" initialAmount="0" />
        <species  id="D" initialAmount="0"  />
    </listOfSpecies>
    <listOfParameters>
      <parameter id="k1" value="4"/>
    </listOfParameters>
    <listOfReactions>
        <reaction id="R1" reversible="false">
            <listOfReactants>
                <speciesReference species="A"/>
                <speciesReference species="B" stoichiometry="2"/>
            </listOfReactants>
            <listOfProducts>
                <speciesReference species="C"/>
                <speciesReference species="D"/>
            </listOfProducts>
            <kineticLaw>
              <math>
                <apply>
                  <times/>
                  <ci> k1 </ci>
                  <ci> A </ci>
                  <apply>
                    <power/>
                    <ci> B </ci>
                    <cn type="integer"> 2 </cn>
                  </apply>
                </apply>
              </math>
            </kineticLaw>
        </reaction>
      </listOfReactions>
    </model>
</sbml>
```

Supponendo che "`model`" sia il modello che stiamo analizzando con la libreria, è possibile, ad esempio, utilizzare i seguenti metodi per estrarre informazioni riguardo il numero di specie presenti nel sistema:

```
model.getNumSpecies()
```

oppure l'id associato all'i-esimo elemento della lista delle reazioni (cioè l'i-esima reazione):

```
model.getReaction(i).getId()
```

Questa libreria, quindi, offre funzioni che permettono di leggere da un modello solo le informazioni a cui si è interessati e permette, inoltre, di modificarlo. Noi l'abbiamo usata esclusivamente per la lettura, mentre per modificare gli attributi del modello abbiamo utilizzato la libreria *libRoadRunner*.

libRoadRunner è una libreria open-source, disponibile per diversi sistemi operativi, inclusi Linux, Windows e Mac OS. Anch'essa ci consente di estrarre

Specie	Parametro
id	id
compartment	value
initialAmount	units
initialConcentration	constant
substanceUnits	
hasOnlySubstanceUnits	
boundaryCondition	
constant	
conversionFactor	

Tabella 3.1: Attributi degli elementi di tipo Specie e di tipo Parametro.

alcune informazioni da un modello e di modificarle [1]. In questo, si differenzia da libSBML in quanto permette di ottenere e modificare solo alcune delle informazioni che, invece, è possibile estrarre o cambiare con libSBML. Permette, inoltre, di ricavare informazioni che non sono direttamente memorizzate nel modello, ma sono calcolabili da quelle a disposizione. Un esempio di informazione che non compare in modo esplicito è la velocità di una reazione che può essere, però, calcolata conoscendo i dati di quest'ultima e delle specie che vi prendono parte. libRoadRunner permette anche di simulare un modello offrendo vari solver per consentire un'analisi utilizzando un approccio deterministico oppure stocastico. Il solver da noi usato è quello deterministico di default: *cvode* [1].

Oltre alle due librerie appena descritte, sono state usate anche le librerie *os*, *shutil* e *pathlib*, che permettono di gestire ed analizzare cartelle e file, e *numpy* e *collections*, per poter lavorare con array e matrici e con collezioni ordinate. Ad esempio, possono essere usate per creare nuove cartelle, spostare file tra queste, modificare array e matrici.

Simulazioni e Petri nets

Siamo interessati a memorizzare due tipi di informazioni generate durante l'esecuzione del programma: le Petri nets dei modelli e i risultati delle loro simulazioni.

Le Petri nets vengono descritte utilizzando il linguaggio *DOT*, un linguaggio apposito per la descrizione di grafi [21], che mediante una sintassi dettagliata è utile per rappresentare grafi orientati e non orientati.

Graficamente, permette di rappresentare nodi di tipo diverso rendendo disponibili varie forme come, ad esempio, cerchi e rettangoli, utilizzati per i posti e le transizioni delle Petri nets. Inoltre, è possibile distinguere diversi tipi di archi [18] e associare un'etichetta a ogni nodo o arco [21]. La Figura 3.3 mostra la rappresentazione in linguaggio DOT della Petri net in Figura 2.5. Qui si possono individuare i posti, definiti con forma circolare (`shape=circle`), e le transizioni, definite, invece, come rettangoli (`shape=box`). Viene usato il simbolo "`->`" per aggiungere un arco, la cui forma può essere successivamente

Figura 3.3: Esempio di un Petri net descritta tramite il linguaggio DOT.

```
digraph "model_name"{
    x [shape=circle];
    y [shape=circle];
    z [shape=circle];
    x_Formation [shape=box];
    x_Formation -> x [arrowhead=vee, label=1];
    z -> x_Formation [arrowhead=tee];
    x_Consumption [shape=box];
    x -> x_Consumption [arrowhead=vee, label=1];
    z -> x_Consumption [arrowhead=dot];
    y_Formation [shape=box];
    y_Formation -> y [arrowhead=vee, label=1];
    x -> y_Formation [arrowhead=dot];
    y_Consumption [shape=box];
    y -> y_Consumption [arrowhead=vee, label=1];
    z_Formation [shape=box];
    z_Formation -> z [arrowhead=vee, label=1];
    y -> z_Formation [arrowhead=dot];
    z_Consumption [shape=box];
    z -> z_Consumption [arrowhead=vee, label=1];
    x -> z_Consumption [arrowhead=tee];
}
```

modificata specificando testa, coda o entrambe. In figura si possono osservare tutti i tipi di arco usati per le Petri nets, i quali si differenziano per la testa: `vee`, utilizzata per i reagenti e i prodotti, `tee`, per gli inibitori, e `dot`, per i promotori. Mediante il campo `label`, è possibile associare un'etichetta ai nodi e agli archi, e, in questo caso, questa rappresenta rispettivamente l'id del nodo e il coefficiente stochiometrico con cui esso partecipa alla reazione. In questo caso si può notare l'etichetta di valore "1" (`label=1`) associata agli archi con `arrowhead=vee`.

Per memorizzare le Petri nets generate abbiamo usato file con estensione `.gv`, tipica per dati rappresentati in linguaggio DOT e preferita all'estensione `.dot` in quanto quest'ultima veniva utilizzata da Microsoft Word [21]. Per salvare i risultati delle simulazioni dei modelli analizzati abbiamo, invece, utilizzato file con estensione `.txt`.

L'utilizzo delle librerie e il contenuto dei file creati saranno illustrati in modo più approfondito nel Capitolo 4.

Capitolo 4

Implementazione

Il software realizzato ha lo scopo di analizzare e simulare, ovvero emulare, il comportamento di reti di reazioni e di creare le rispettive Petri nets. I modelli utili al nostro scopo sono quelli che presentano alcune caratteristiche ben definite.

Per esaminare in batteria i modelli, il software esegue i seguenti passi:

1. selezione dei modelli di interesse;
2. verifica della struttura del modello utilizzando i teoremi di Feinberg;
3. simulazione dei modelli;
4. costruzione delle Petri nets.

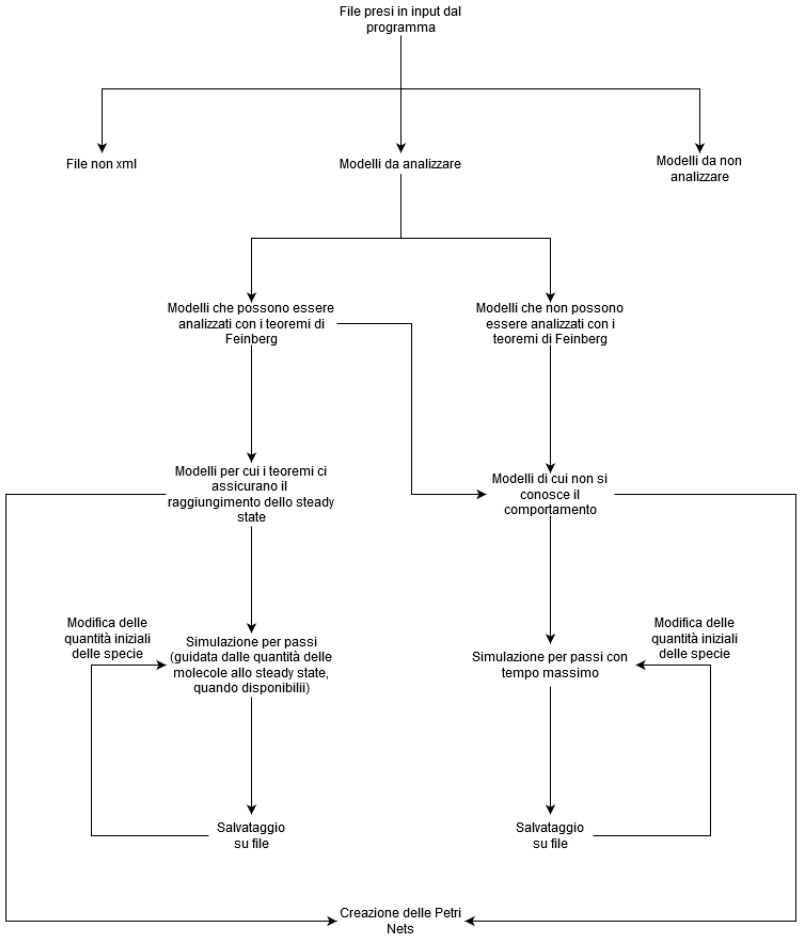
Nella Figura 4.1 è rappresentato in modo schematico l'esecuzione del programma creato.

Il software è stato suddiviso in moduli, ognuno caratterizzato da una propria funzionalità. A loro volta i moduli sono stati suddivisi in funzioni. Questa scelta permette di utilizzare solo le funzionalità a cui si è interessati senza doverle necessariamente eseguire tutte, di aumentare la leggibilità del codice e di modificare più facilmente.

Il progetto è suddiviso in 6 moduli:

- `BaseFunction`, che offre funzioni che si occupano della gestione delle eccezioni di alcuni metodi delle librerie usate;
- `SelectFiles`, che permette di filtrare i modelli in base alle loro caratteristiche;
- `Deficiency_Calculation`, per analizzare il comportamento del sistema utilizzando i teoremi di Feinberg;
- `Simulation`, per la simulazione di uno o più modelli in base alle informazioni che si hanno su di essi. Inoltre permette di distinguere i modelli che sono descritti mediante la legge di azione di massa;

Figura 4.1: Schema sul funzionamento del programma.



- `PetriNets`, per creare e salvare su file la Petri net associata al modello;
- `Main`, per definire il flusso del programma.

Ogni modulo verrà descritto in dettaglio nei paragrafi successivi.

Dopo aver definito i moduli, viene creato il package pronto per l'installazione attraverso la creazione di file e script Python aggiuntivi.

4.1 Modulo 1: BaseFunction

Nel modulo BaseFunction sono state definite funzioni che si occupano della gestione delle eccezioni di alcuni metodi delle librerie usate. È stato scelto di creare un modulo apposito in quanto questi metodi vengono utilizzati più volte

in differenti moduli e non fanno parte della specifica funzionalità di nessuno di essi.

Le funzioni offerte sono 4 e permettono di gestire in modo controllato la rimozione di file, la creazione di directory e la richiesta all'utente dell'inserimento di un numero intero. Le prime due, osservabili nella Figura 4.2, si occupano della creazione di una directory. Per fare ciò viene eseguito un controllo circa l'esistenza della cartella indicata in input che, nel caso in cui non esista, viene creata. La prima funzione differisce dalla seconda in quanto, in caso di errore durante la creazione, provoca la terminazione del programma, mentre l'altra indica il successo o il fallimento dell'operazione restituendo un codice numerico (rispettivamente 0 e 1) delegando la gestione del fallimento alla funzione chiamante. Si può osservare l'uso delle librerie `pathlib`, della quale viene usato il modulo `Path`, e `os` rispettivamente per controllare se la directory indicata esista già e, in caso contrario, per crearla.

Figura 4.2: Funzioni per la creazione di una cartella.

```
5   def createDirectoryExit(path):
6       if not Path(path).is_dir():
7           try:
8               os.mkdir(path)
9           except:
10              exit("An error occurred during creation of directory")


13  def createDirectory(path):
14      if not Path(path).is_dir():
15          try:
16              os.mkdir(path)
17          except:
18              print("An error occurred during creation of directory")
19              return 1
20      return 0
```

Un'altra funzione che compare in questo modulo si occupa della rimozione di un file e presenta la seguente segnatura: `removeFile(path)`, dove `path` indica il file da rimuovere. Per fare ciò, in modo simile a come era stato fatto per le directory, viene controllato se l'argomento corrisponde ad un file esistente. In questo caso, si procede con la sua cancellazione. Viene restituito un codice numerico che corrisponde a 0 nel caso in cui la rimozione abbia avuto successo, 1 altrimenti.

Infine, in Figura 4.3 si può osservare un'ultima funzione, la quale permette di gestire gli argomenti inseriti da linea di comando che presentano il vincolo di essere numeri interi. Nel caso in cui i valori possibili siano limitati e possano essere elencati, è possibile specificarli attraverso l'uso di una lista. In questo caso viene controllato se il valore in input corrisponde a uno di essi, ossia se è presente nella lista dei possibili valori. Se ciò non accade o se l'input non è un numero intero, la funzione provoca la terminazione del programma, altri-

Figura 4.3: Funzione per l'input di un valore intero da riga di comando.

```
32  def numericalInput(outputString, allowValues):
33      try:
34          choice=int(input(outputString))
35      except ValueError:
36          exit("Error: insert a number")
37      if allowValues!=[] and choice not in allowValues:
38          exit("Error: invalid choice")
39      return choice
```

menti ritorna il valore inserito dall'utente. Questa funzione verrà utilizzata per ottenere dall'utente le informazioni necessarie all'esecuzione delle simulazioni.

4.2 Modulo 2: SelectFiles

SelectFiles è il modulo in cui è implementata la parte di software che permette di selezionare i modelli che presentano le seguenti caratteristiche:

- sono non qualitativi;
- non presentano né regole né eventi.

Per poterli identificare vengono utilizzate le librerie libRoadRunner e libSBML. Per utilizzare la prima vengono chiamate le funzioni seguenti:

```
rr=roadrunner.RoadRunner()
rr.load(file)
```

rispettivamente per creare un riferimento a libRoadRunner e caricare il modello contenuto nel file indicato. Per poter analizzare un modello con libSBML, questo viene letto grazie alla successiva composizione di alcuni dei metodi offerti:

```
model =
libsbml.SBMLReader().readSBMLFromString(rr.getSBML()).getModel()
```

che permettono così di creare un oggetto di tipo Model che rappresenta il modello da studiare.

I modelli qualitativi, per essere descritti, usano un'estensione dell'SBML e per questo motivo possono essere individuati mediante il seguente controllo:

```
model.getPlugin('qual')!=None
```

dove il metodo `getPlugin` ritorna l'oggetto corrispondente all'interfaccia estesa del plug-in specificato. Se il modello non è qualitativo allora il plug-in 'qual', e di conseguenza l'oggetto corrispondente, non è definito e la funzione ritorna 'None' [13]. Quindi se il modello è qualitativo questo controllo risulta vero.

In una CRN sono presenti regole o eventi se questi sono in numero maggiore di 0. Ciò può essere verificato utilizzando le apposite funzioni offerte da libSBML:

```
model.getNumRules()>0 or model.getNumEvents()>0
```

dove la prima restituisce il numero di regole definite nel modello, mentre la seconda il numero di eventi.

Alcuni modelli non possono essere letti oppure vengono letti in modo errato, per numero di specie, reazioni o entrambe, da libRoadRunner. Nel primo caso, viene sollevata un'eccezione al momento del caricamento del file da parte della libreria. Per riconoscere invece modelli che ricadono nel secondo caso viene eseguito un confronto tra i dati ottenuti con la libreria libSBML e con libRoadRunner. Supponendo che le due librerie leggano lo stesso modello, viene ricavato da entrambe il numero di specie e reazioni e vengono confrontati i valori ottenuti. Nel caso in cui almeno una delle due quantità non coincida allora è presente un errore nella lettura del modello. Per fare ciò viene utilizzato il seguente controllo:

```
model.getNumSpecies()!=rr.model.getNumFloatingSpecies() or
    model.getNumReactions()!=rr.model.getNumReactions()
```

che risulta vero in caso di errori, falso altrimenti.

Un modello che non può essere letto oppure che viene letto in modo errato, non viene selezionato per l'analisi e la simulazione perché, nel primo caso, non è possibile eseguirla a causa dell'errore sollevato dalla libreria libRoadRunner, mentre, nel secondo, i risultati sarebbero inesatti in quanto non verrebbero presi in considerazione tutti i fenomeni presenti nel sistema.

Le funzioni definite in questo modulo sono due: `selectOneModel` e `select`.

La prima esamina un modello cercando una delle caratteristiche sopra indicate e restituisce alla funzione chiamante un codice numerico che indica se il modello dev'essere simulato oppure no e, in quest'ultimo caso, il codice numerico specifica il motivo del rifiuto.

In Figura 4.4 è possibile osservare l'implementazione della funzione, la quale prende in input 3 argomenti: `rr`, che è il riferimento alla libreria libRoadRunner, `path`, che rappresenta il percorso dove cercare il modello, `file`, che corrisponde al file cercato. Innanzitutto, viene controllato se il file specificato ha estensione `.xml` e, in questo caso, il modello viene caricato dalla libreria libRoadRunner. In caso di errore la funzione termina restituendo l'apposito codice. Altrimenti viene creato l'oggetto di tipo Model, leggendo il modello con la libreria libSBML, e viene controllato se una delle condizioni sopra descritte è verificata. In questo caso la funzione restituisce 1, altrimenti 0.

Il metodo `select` ha un comportamento simile, ma gli argomenti presi in input sono differenti: `rr`, `dirToExamine`, che corrisponde alla cartella contenente i file da analizzare, `dirNotAnalyze`, che indica in quale cartella spostare i modelli a cui non si è interessati, `others`, che rappresenta la cartella dove spostare i file non xml. Le ultime due, nel caso in cui non esistano, vengono create attraverso le funzioni definite nella Sezione 4.1. La funzione si differenzia dalla precedente in quanto studia tutti i file presenti nella directory indicata e non presenta valori di ritorno, bensì separa dagli altri i modelli che presentano le caratteristiche sopra citate e i file che non presentano l'estensione `.xml` spostandoli in cartelle

Figura 4.4: Funzione per analizzare le caratteristiche di un modello.

```
48  def selectOneModel(rr, path, file):
49      if file.endswith('.xml'):
50          try:
51              #il modello viene letto con libRoadRunner
52              rr.load(path+"\\"+file)
53          except:
54              print('An error occurs during file load: To not analyze')
55              return -1
56
57          #il modello viene letto con libSBML
58          model = libsbml.SBMLReader().readSBMLFromString(rr.getSBML()).getModel()
59
60          if  model.getNumRules()>0  or model.getNumEvents()>0 or model.getPlugin('qual')!=None or \
61                  model.getNumSpecies()!=rr.model.getNumFloatingSpecies() or \
62                  model.getNumReactions()!=rr.model.getNumReactions():
63              print('To not analyze')
64              return 1
65
66          #è un modello da analizzare
67          return 0
68      else:
69          #non è un file con estensione .xml
70          return 2
```

specificate. Per eseguire lo spostamento viene utilizzata la libreria `shutil` e, più in particolare, il comando `move`:

```
shutil.move(file, destination_path)
```

4.3 Modulo 3: Deficiency_Calculation

Per i modelli che raggiungono lo stato stazionario si vorrebbe usare una simulazione che tiene conto di questa informazione. Come detto nel Paragrafo 2.3, è possibile individuare l'andamento di alcuni sistemi studiandone la struttura senza aver bisogno di simularli nelle diverse configurazioni iniziali. Per fare ciò vengono utilizzati i teoremi di Feinberg che, per modelli con determinate caratteristiche, assicurano il raggiungimento dello stato stazionario. Per applicare questi teoremi a un modello, nel modulo Deficiency_Calculation è stata implementata una funzione per il calcolo della deficiency e il controllo della proprietà di reversible o weakly reversible della rete supponendo di analizzare solo modelli descritti con la legge di azione di massa.

Nel sottoparagrafo 4.3.1 verrà descritto come determinare le informazioni necessarie al calcolo della deficiency, mentre nel sottoparagrafo 4.3.2 verrà mostrata la verifica della proprietà di reversible o weakly reversible di una rete.

4.3.1 Calcolo della deficiency

Come descritto nel Capitolo 2.3, per calcolare la deficiency di un modello sono necessarie tre informazioni: il numero di nodi, il numero di linkage classes e il rango della matrice stochiometrica. Per calcolarle viene analizzato il grafo

corrispondente al modello considerato utilizzando la struttura dati ritenuta più appropriata. È stato scelto di rappresentare il grafo (V, E) tramite liste di adiacenza, preferite alla matrice di incidenza perché i grafi associati alle CRN nella maggior parte dei casi sono sparsi, ossia il numero di archi è molto minore rispetto al numero di nodi al quadrato [19]. Per questo motivo, l'uso delle liste di adiacenza permette una rappresentazione più compatta, con un'occupazione di memoria minore rispetto alla matrice di incidenza. La memoria occupata con la struttura dati scelta è pari al numero di archi, mentre con l'altra è pari al numero di nodi per il numero di archi. Se il grafo è sparso allora

$$\#E << \#V * \#E \tag{4.1}$$

Le liste di adiacenza definite vengono usate per calcolare il numero di nodi e il numero di linkage classes del grafo. Successivamente viene costruita la matrice stochiometrica di cui interessa il rango.

Rappresentazione del grafo e numero di nodi

Per creare le liste di adiacenza viene mantenuta in memoria una lista, `nodes`, che contiene a sua volta una lista per ogni nodo visto fino a un certo istante e viene usata per controllare se un certo nodo viene incontrato per la prima volta oppure no. Ogni elemento di queste liste corrisponde a una stringa rappresentante una specie che compare nel nodo considerato e il coefficiente stochiometrico ad essa associato. L'ordine in cui compaiono queste liste viene usato per associare un nodo alla propria lista di adiacenza. Più precisamente, la posizione di una lista in `nodes` corrisponde all'indice della lista di adiacenza che lo rappresenta. Dato che viene definita una lista di adiacenza per ogni nodo, queste sono in numero #V.

Per creare i nodi vengono analizzate tutte le reazioni, una alla volta, mediante un ciclo `for`. Per ricavare il numero di reazioni della CRN viene usato il seguente comando della libreria libSBML:

```
model.getNumReactions()
```

il quale considera tutte le reazioni definite in modo esplicito, ma non tiene conto delle reazioni inverse [17] i cui archi vanno, però, aggiunti alle liste di adiacenza.

Supponiamo di analizzare l'i-esima reazione. Per definire i corrispondenti nodi del grafo bisogna identificare i reagenti e i prodotti.

Per creare il nodo dei reagenti, le informazioni necessarie sono le specie che compaiono nel nodo e i loro coefficienti stochiometrici. Possono essere estratte dal modello ottenendo, innanzitutto, la lista dei reagenti della reazione con la specifica funzione implementata nella libreria libSBML [17]:

```
model.getListOfReactions().get(i).getListOfReactants()
```

Poi, per ognuno di questi reagenti, è possibile ricavare l' id della specie reagente `r` e il suo coefficiente stochiometrico con i successivi metodi [17]:

Figura 4.5: Creazione di un nodo contente i reagenti.

```
53          #viene costruito il nodo composto dai reagenti
54          nodeReactant=[]
55          for r in model.getListOfReactions().get(i).getListOfReactants():
56              nodeReactant.append(str(r.getStoichiometry())+str(r.getSpecies()))
57          if len(nodeReactant)==0:
58              nodeReactant.append("0")
59          else:
60              nodeReactant.sort()
```

`r.getSpecies()`
`r.getStoichiometry()`

Avendo queste informazioni per ogni reagente, il nodo viene costruito inserendole nella lista corrispondente sotto forma di stringhe. Prendendo come esempio la seguente reazione chimica:

$$A + 2B \xrightarrow{k_1} C + D \tag{4.2}$$

per prima cosa viene ottenuta la lista dei reagenti [A, 2B] e ognuno di essi viene analizzato singolarmente. La lista che corrisponde al nodo, supponiamo `nodeReactant`, inizialmente è vuota. Analizzando il primo reagente, A, la stinga corrispondente sarà così costruita:

“1A”

Essa viene aggiunta alla lista del nodo. Analizzando il secondo reagente, alla lista `nodeReactant` viene aggiunto l’elemento corrispondente al secondo reagente e al suo coefficiente stochiometrico:

“2B”

Dopo aver analizzato tutti i reagenti, la lista corrispondente al nodo viene ordinata. Quindi il nodo costruito verrà rappresentato nel seguente modo:

nodeReactant=[“1A”, “2B”]

Nel caso in cui la reazione non abbia reagenti perché essa avviene in modo spontaneo, la sua lista dei reagenti è vuota. Pertanto il nodo sarà composto dalla specie fittizia “0”:

nodeReactant=[“0”]

La creazione del nodo che contiene i prodotti della reazione avviene in modo analogo. Viene analizzata la lista dei prodotti ottenuta con la specifica funzione di libSBML [17]:

`model.getListOfReactions().get(i).getListOfProducts()`

Figura 4.6: Ricerca di un nodo nella lista dei nodi visti in precedenza.

```
76          #per ricordare l'indice del nodo nell'array nodes
77          indexNodeReactant=-1
78          j=0
79          while indexNodeReactant==-1 and j<len(nodes):
80              #se sono uguali
81              if nodeReactant==nodes[j]:
82                  indexNodeReactant=j
83              #se sono diversi
84              else:
85                  j=j+1
```

In Figura 4.5 è rappresentata l'implementazione del procedimento appena descritto per la costruzione del nodo contenente i reagenti.

Una volta che un nodo è stato creato bisogna verificare se questo è già stato incontrato in precedenza oppure no. Per fare ciò viene utilizzata la lista `nodes` creata all'inizio. Viene confrontato il nodo costruito con ogni elemento della lista interrompendo la ricerca nel caso in cui esso venga trovato. Per confrontare due nodi vengono comparate le due liste che li compongono. Se queste sono uguali allora essi sono uguali, altrimenti sono diversi.

In Figura 4.6 è mostrata la ricerca di un nodo nella lista `nodes` e della sua posizione al suo interno.

Dopo aver creato i nodi dei reagenti e dei prodotti della reazione studiata, se la reazione è irreversibile, alla lista di adiacenza del primo viene aggiunto l'indice del secondo:

```
adjacency_list[indexNodeReactant].append(indexNodeProduct)
```

Altrimenti l'indice di ognuno dei due nodi viene aggiunto alla lista di adiacenza dell'altro. Per verificare la reversibilità viene usata la seguente funzione di libSBML:

```
getReversible()
```

Considerando la rete di reazioni della Tabella 2.2, notiamo che le sue liste di adiacenza, costruite con il procedimento definito in precedenza, saranno definite come mostrato in Figura 4.7.
Qui si può osservare, ad esempio, che la specie `species_0` ha due archi uscenti: l'arco verso la specie di indice 1, ovvero `species_1` e l'arco verso quella di indice 5, ossia `species_4`.

Il numero di nodi della rete è pari alla dimensione della lista `nodes` oppure al numero di liste di adiacenza ottenute dopo aver analizzato tutte le reazioni della rete. A questo punto si ha a disposizione la prima informazione necessaria a calcolare la deficiency.

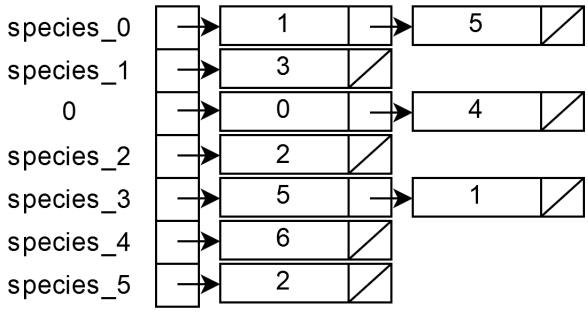


Figura 4.7: Liste di adiacenza rappresentanti un grafo.

Numero di linkage classes

Per calcolare il numero di linkage classes vengono analizzate tutte le liste di adiacenza create. Interessa scoprire, partendo da un nodo, quanti e quali altri nodi possono essere raggiunti o possono raggiungere il nodo seguendo la direzione degli archi del grafo. Se da un nodo non è possibile arrivare a un altro e viceversa allora i due appartengono a componenti connesse diverse. Per calcolare il numero di queste ultime viene utilizzata una lista di supporto che contiene gli indici delle liste di adiacenza e il loro ordine in questa lista definisce l'ordine con cui queste ultime vengono esaminate. Questa nuova lista all'inizio è composta dagli indici in ordine crescente, ma i suoi elementi possono essere scambiati durante lo studio delle componenti connesse. Viene usata per ricordare quali sono i nodi già assegnati ad una linkage class ad un determinato istante e quali, invece, non sono stati ancora assegnati. Per fare ciò, viene definita una variabile (supponiamo `last`) che si comporta da limite suddividendo il primo gruppo dal secondo, puntando all'ultimo elemento assegnato a una componente connessa. Per tutti gli elementi che si trovano a sinistra di quello puntato da `last` è già stata individuata la linkage class, mentre per quelli a destra dev'essere ancora trovata. Viene definito un contatore per memorizzare il numero di linkage classes trovate.

Per calcolare il numero di componenti connesse viene utilizzato il seguente procedimento:

1. viene costruita la lista di supporto che contiene i valori compresi da 0 al numero di nodi (escluso) in ordine crescente;

2. si scorre la lista appena creata fermandosi se `last` assume valore pari al numero di nodi meno 1 perchè ciò indica che è stata individuata la linkage class di ogni nodo;

3. a ogni iterazione, se questa variabile ha valore inferiore rispetto all'indice del ciclo che scorre la lista creata, allora il nodo che stiamo analizzando non appartiene a nessuna componente connessa già vista;

Figura 4.8: Implementazione dell'algoritmo per il calcolo delle linkage classes.

```
146         indexes=list(range(np.shape(adjacency_list)[0]))
147         numLinkageC=0
148         last=-1
149         j=0
150         while j<len(adjacency_list) and last<len(adjacency_list)-1:
151             #se la reazione non fa parte di nessuna linkage classes
152             if last<j:
153                 numLinkageC=numLinkageC+1
154                 last=last+1
155             i=0
156             while i<len(adjacency_list[indexes[j]]) and \
157                 last<len(adjacency_list)-1:
158                 if indexes.index(adjacency_list[indexes[j]][i])>last:
159                     last=last+1
160                     pos=indexes.index(adjacency_list[indexes[j]][i])
161                     tmp=indexes[last]
162                     indexes[last]= indexes[pos]
163                     indexes[pos]=tmp
164                 i=i+1
165             i=last+1
166             while i<len(adjacency_list) and last<len(adjacency_list)-1:
167                 if indexes[j] in adjacency_list[indexes[i]]:
168                     last=last+1
169                     tmp=indexes[last]
170                     indexes[last]= indexes[i]
171                     indexes[i]=tmp
172                 i=i+1
173             j=j+1
```

4. durante l'analisi di un particolare nodo, viene scorsa la sua lista di adiacenza per vedere quali nodi fanno parte della stessa componente;

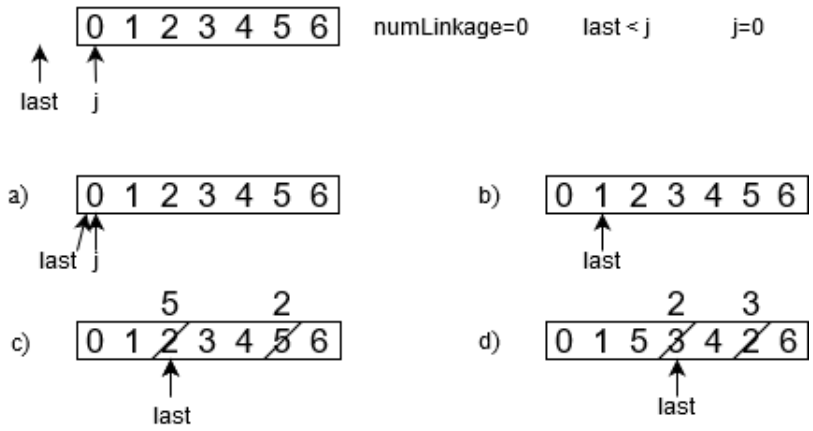
5. viene, infine, controllato, per ogni nodo non assegnato a nessuna linkage class, se le loro liste di adiacenza contengono il nodo studiato perché, in questo caso, la componente connessa è la stessa.

Alla fine, il numero di linkage classes è contenuto nella variabile definita in precedenza.

In Figura 4.8 è mostrata l'implementazione del procedimento appena descritto, dove `index` rappresenta la lista di supporto che stabilisce l'ordine di studio delle liste di adiacenza e `numLinkageC` corrisponde al contatore che contiene il numero di componenti connesse.

Consideriamo il grafo rappresentato dalle liste di adiacenza presenti in Figura 4.7. È possibile osservarne la lista creata per calcolare il numero di componenti

Figura 4.9: Configurazione iniziale e analisi del primo nodo per il calcolo delle linkage classes.



connesse e l'analisi della prima lista di adiacenza in Figura 4.9. I nodi del grafo sono 7 e ognuno di essi viene rappresentato con l'indice della propria lista di adiacenza. Il primo nodo analizzato è il nodo presente in posizione j=0, ossia quello di indice 0 corrispondente a "species_0". Viene scorsa la sua lista di adiacenza e viene trovato il nodo di indice 1 che viene spostato nella posizione successiva a 0 nella lista di supporto incrementando la variabile `last` di 1. In questo caso la nuova posizione corrisponde alla posizione attuale. In questo modo è stata ottenuta la configurazione 'b'. Successivamente viene trovato il nodo di indice 5 che viene spostato dopo l'ultimo nodo visto dalle iterazioni precedenti, ossia quello di indice 1. Per fare ciò viene scambiato con l'elemento che attualmente occupa quella posizione, ovvero il numero 2. Dopo questa operazione si è giunti alla configurazione 'c' in Figura 4.9. Dopo aver esaminato la lista di adiacenza del nodo di indice 0, questo indice viene cercato nelle liste dei nodi non ancora incontrati. Questo viene trovato nella lista del nodo "0" il quale ha indice 2. L'elemento 2 nella lista di supporto viene scambiato con l'elemento successivo a quello puntato da `last` incrementando quest'ultima variabile. In questo modo il 2 prende il posto del 3, come mostrato dalla configurazione 'd' rappresentata in figura. L'indice 0 non viene trovato in nessun'altra lista di adiacenza. Si può notare che gli elementi a sinistra e quello puntato dalla variabile `last` sono stati assegnati ad una linkage class, in questo caso la stessa, e che per quelli a destra, invece, la componente connessa è sconosciuta. Successivamente viene analizzato il nodo successivo presente nella lista di supporto, ovvero quello di indice 1, eseguendo un procedimento analogo.

Matrice stochiometrica

L'ultima informazione necessaria per il calcolo della deficiency è il rango della matrice stochiometrica. Come descritto nel Capitolo 2.3, il numero di righe e

```
217       #creazione della matrice di 0
218       stochiometricMatrix=np.zeros((rr.model.getNumReactions(), \
219           rr.model.getNumFloatingSpecies()))
220       #viene analizzata ogni reazione
221       for i in range(model.getNumReactions()):
222           #vettore del nodo dei reagenti
223           vReactant=np.zeros(rr.model.getNumFloatingSpecies())
224           for r in model.getListOfReactions().get(i).getListOfReactants():
225               #nella posizione corrispondente alla specie viene inserito il suo
226               #coefficiente stochiometrico
227               vReactant[rr.model.getFloatingSpeciesIds().index(r.getSpecies())]=\
228                   r.getStoichiometry()
229           #vettore del nodo dei prodotti
230           vProduct=np.zeros(rr.model.getNumFloatingSpecies())
231           for r in model.getListOfReactions().get(i).getListOfProducts():
232               #nella posizione corrispondente alla specie viene inserito il suo
233               #coefficiente stochiometrico
234               vProduct[rr.model.getFloatingSpeciesIds().index(r.getSpecies())]=\
235                   r.getStoichiometry()
236           #gli elementi dei due vettori vengono sottratti e la matrice
237           #viene aggiornata di conseguenza
238           for j in range(rr.model.getNumFloatingSpecies()):
239               stochiometricMatrix[i][j]=vProduct[j]-vReactant[j]
```

Figura 4.10: Creazione della matrice stochiometrica di una CRN.

il numero di colonne della matrice stochiometrica, associata a una CRN, sono uguali rispettivamente al numero di reazioni e al numero di specie che compaiono nella rete. La matrice stochiometrica che viene creata per il calcolo del rango non tiene, però, conto delle reazioni inverse. Queste non influiscono sul risultato atteso perché i vettori corrispondenti a una reazione e alla sua inversa sono linearmente dipendenti in quanto uno è uguale all'altro moltiplicato per -1. Ad esempio, considerando la reazione chimica 2.3, i vettori corrispondenti vengono calcolati nel seguente modo:

$$\begin{matrix}(C+D)-(A+2B)\\(A+2B)-(C+D)\end{matrix} \tag{4.3}$$

Sappiamo, però, che

$$(A+2B)-(C+D)=-((C+D)-(A+2B)) \tag{4.4}$$

Quindi calcolando il primo vettore è possibile ottenere il secondo moltiplicando il primo per -1:

$$\begin{matrix}(C+D)-(A+2B)=[0,0,1,1]-[1,2,0,0]=[-1,-2,1,1]\\(A+2B)-(C+D)=-1*[-1,-2,1,1]=[1,2,-1,-1]\end{matrix} \tag{4.5}$$

Detto ciò, per costruire la matrice per prima cosa ne viene creata una composta da soli 0 di dimensione pari al numero di reazioni (non contando le inverse)

per il numero di specie utilizzando la libreria `numpy` e, più precisamente, il suo metodo `zeros`:

```
np.zeros((rr.model.getNumReactions(), rr.model.getNumFloatingSpecies()))
```

Successivamente per ogni reazione considerata vengono creati i vettori corrispondenti ai due nodi. Questi vengono inizialmente definiti come vettori di zeri e i loro valori vengono aggiornati analizzando i reagenti o i prodotti, estraendo i coefficienti stochiometrici loro associati utilizzando la funzione apposita, della libreria libSBML, vista in precedenza durante la costruzione dei nodi della rete. Dopo aver creato i due vettori, questi vengono sottratti tra loro e la matrice viene aggiornata di conseguenza.

In Figura 4.10 è mostrata l'implementazione del processo appena descritto.

A questo punto, avendo costruito la matrice stochiometrica è possibile calcolarne il rango utilizzando l'apposita funzione della libreria `numpy`:

```
np.linalg.matrix_rank(stochiometricMatrix)
```

È ora possibile calcolare la deficiency mediante l'equazione 2.9, avendo tutte le informazioni necessarie.

4.3.2 Proprietà di reversibility e weakly reversibility della rete

La seconda proprietà della CRN che bisogna analizzare è quella di reversibility o weakly reversibility. La prima è facilmente verificabile in quanto una rete è reversible se per ogni reazione è presente anche l'inversa.

Nella Figura 4.11 è mostrato il codice per eseguire questo controllo e viene usata una variabile, `allReversible`, per memorizzarne l'esito. Si itera sulle reazioni controllandone la reversibilità chiamando il metodo implementato nella libreria libSBML:

```
getReversible()
```

che restituisce "True" nel caso in cui la reazione sia reversibile, "False" altrimenti.

```
264     allReversible=True
265     for reaction in model.getListOfReactions():
266         if reaction.getReversible()==False:
267             allReversible=False
268             break
```

Figura 4.11: Implementazione dell'analisi della proprietà di reversible di una rete di reazioni.

Nel momento in cui viene trovata una reazione non reversibile, il controllo viene interrotto in quanto la proprietà non è verificata. Se ciò accade viene controllata la validità della seconda proprietà ovvero viene studiato se la rete è weakly revesible.

Per eseguire questo controllo vengono analizzate tutte le liste di adiacenza così da ottenere per ogni nodo tutti i nodi raggiungibili da esso. Per fare ciò vengono utilizzate due liste di supporto. La prima viene usata per ricordare quali sono i nodi incontrati così da ispezionare ogni nodo una sola volta. Invece, la seconda è composta da altre liste, una per ogni nodo, e in queste vengono memorizzati i nodi raggiungibili dal nodo associato alla lista. Per costruirle si itera sui nodi finché non sono stati esaminati tutti utilizzando una *visita in profondità ricorsiva* per esplorare il grafo. Nella visita in profondità ricorsiva, ogni volta che da un nodo se ne incontra un altro si passa a ispezionare quest'ultimo. Quando la sua ispezione termina si torna all'ispezione di quello precedente lasciata in sospeso [19]. In Figura 4.12 è mostrata la creazione delle due liste e l'iterazione sui nodi facendo attenzione a non esplorare lo stesso nodo due volte.

```
247         #lista dei nodi esplorati
248         explored=[]
249         #lista per la raggiungibilità dei nodi
250         reachables=[]
251         for i in range(np.shape(adjacency_list)[0]):
252             reachables.append([])
253 
254         i=0
255         while len(explored)<np.shape(adjacency_list)[0]:
256             #se l'i-esimo nodo non è stato già esplorato
257             if i not in explored:
258                 recursive_visit(rr, i, explored, reachables, adjacency_list)
259             i=i+1
```

Figura 4.12: Creazione delle liste e chiamata della visita ricorsiva.

Quando viene chiamata la funzione ricorsiva per l'ispezione di un nodo, questo viene aggiunto alla lista degli esplorati. Ogni elemento della sua lista di adiacenza e le liste dei nodi raggiungibili corrispondenti vengono aggiunti alla sua lista dei nodi raggiungibili eliminando eventuali duplicati. Nel caso in cui i nodi trovati non siano stati ancora visitati, viene chiamata la funzione ricorsiva su di essi, mostrata in Figura 4.13.

Dopo aver visitato in questo modo tutti i nodi, vengono aggiornate le liste che sono state costruite in questo modo considerando, per ognuna di esse, quelle di tutti i nodi raggiungibili. In questo modo, in ogni lista si ottengono tutti i nodi raggiungibili dal determinato nodo. A questo punto, per verificare se una rete è weakly reversible, per ogni nodo viene controllato che questo compaia nella lista che contiene i nodi raggiungibili associate a quelli che compaiono nella sua lista di adiacenza. Se ciò non è verificato allora la proprietà non è soddisfatta.

```
6   def recursive_visit(rr, indexRoot, explored, reachables, adjacency_list):
7       explored.append(indexRoot)
8       for i in range(len(adjacency_list[indexRoot])):
9           reachables[indexRoot]=np.concatenate((reachables[indexRoot],\
10               [adjacency_list[indexRoot][i]]))
11
12          #se il nodo non è già stato esplorato
13          if adjacency_list[indexRoot][i] not in explored:
14              #lo esploro
15              recursive_visit(rr, adjacency_list[indexRoot][i], explored,\
16                   reachables, adjacency_list)
17
18          reachables[indexRoot]=list(dict.fromkeys(np.concatenate((\
19              reachables[indexRoot], reachables[adjacency_list[indexRoot][i]]))))
```

Figura 4.13: Visita ricorsiva del grafo.

La funzione, dopo aver calcolato la deficiency della CRN e averne analizzato le proprietà di reversibility e weakly reversibility, restituisce una lista con queste due informazioni, [deficiency, 0/1], dove il secondo elemento assume valore 0 nel caso in cui valgano le proprietà di reversible o weakly reversible e 1 nel caso in cui non valgano.

4.4 Modulo 4: Simulation

Nel modulo Simulation sono definite le funzioni che si occupano della simulazione dei modelli. È presente, inoltre, una funzione utilizzata per decidere se l'andamento di un certo modello dev'essere necessariamente intuito tramite la sua simulazione oppure se può essere analizzato con i teoremi di Feinberg, definiti nel Paragrafo 2.3. Questa verifica se il modello considerato è descritto mediante la legge di azione di massa, in quanto, se ciò non accade, i teoremi non possono essere applicati. Come illustrato nel Capitolo 2.2, le velocità delle reazioni, studiate con la legge di azione di massa, sono influenzate solo dalle specie reagenti e le loro formule presentano come operazioni aritmetiche solo moltiplicazioni e potenze. Nel caso in cui una reazione sia reversibile, nella descrizione della sua legge cinetica possono comparire anche le specie prodotte e l'operatore di sottrazione, in quanto si tiene conto della velocità della reazione inversa. Per questi motivi viene, innanzitutto, verificata, per ogni reazione del sistema, la presenza di modificatori, ossia specie che ne influenzano la velocità, ma che non agiscono né da reagenti né da prodotti. Per fare ciò viene letto il loro numero mediante il seguente comando della libreria libSBML [13]:

```
model.getReaction(i).getNumModifiers()
```

dove l'indice `i` indica che si sta prendendo in considerazione l'i-esima reazione.

Successivamente viene analizzata la legge cinetica di cui si studiano i nodi, ovvero i suoi componenti e in particolare i suoi operatori aritmetici e le funzioni che vi appaiono. In Figura 4.14 è mostrata la parziale implementazione per

l'analisi di una legge cinetica. Per prima cosa viene creata una lista con gli id delle funzioni definite dall'utente, ovvero degli elementi presenti nel modello appartenenti alla seguente classe:

`FunctionDefinitions`

Non tutti i modelli presentano funzioni definite e, per questo motivo, la lista creata potrebbe essere vuota. Successivamente, per ogni reazione, viene creata un'ulteriore lista contenente i nodi della legge mediante il seguente comando [13]:

```
model.getListOfReactions().get(i).getKineticLaw().
        getMath().getListOfNodes()
```

supponendo ancora di analizzare l'i-esima reazione.

Per ogni nodo rappresentante un'operazione matematica, riconosciuto grazie all'apposita funzione della libreria libSBML [13]:

`isOperator()`,

ne viene confrontato il nome con le stringhe "minus", "times" e "power", rappresentanti rispettivamente sottrazione, moltiplicazione e potenza. Nel caso in cui sia presente un operatore diverso, il modello non è descritto con la legge di azione di massa e quindi non è possibile applicarvi i teoremi sopra citati. Vengono cercati anche nodi corrispondenti a funzioni, come ad esempio al valore assoluto, individuate chiamando la seguente funzione:

`isFunction()`

Questa restituisce anche le funzioni definite dall'utente e non proprie dell'SBML. Per questo motivo viene controllato se le funzioni trovate appartengono alla lista con gli id creata in precedenza. Se è presente una funzione non definita dall'utente, ovvero non appartenente a quest'ultima lista, che non rappresenta

```
419       #creazione della lista con gli id delle funzioni definite nel modello
420       nameFunctions=[f.getId() for f in model.getListOfFunctionDefinitions()]
421
422       #controllo sulle leggi cinetiche
423       for i in range(model.getNumReactions()):
424           l=model.getListOfReactions().get(i).getKineticLaw().getMath().getListOfNodes()
425           for j in range(l.getSize()):
426               if l.get(j).isOperator() and l.get(j).getOperatorName()!="minus" and \
427                   l.get(j).getOperatorName()!="times" and \
428                       l.get(j).getOperatorName()!="power":
429                   return 1
430               #se è presente una funzione diversa da -, *, ^
431               if l.get(j).isFunction() and l.get(j).getName() not in nameFunctions and\
432                    l.get(j).getName()!="minus" and l.get(j).getName()!="times" and\
433                         l.get(j).getName()!="power":
434                   return 1
```

Figura 4.14: Analisi della leggi cinetiche delle reazioni.

sottrazione, moltiplicazione e potenza allora, anche in questo caso, non è possibile applicare i teoremi al modello. Se, invece, ciò non accade, le funzioni definite dall'utente vengono analizzate in modo analogo a come era stato fatto per la legge, cioè ne vengono studiati i nodi ottenuti grazie alla composizione dei comandi seguenti:

```
model.getFunctionDefinition(i).getBody().getListOfNodes()
```

Il metodo che si occupa di studiare le leggi in questo modo restituisce 1 se il modello non rispetta i requisiti per essere analizzato con i teoremi definiti da Feinberg, 0 altrimenti.

Gli altri metodi si occupano della gestione delle simulazioni. I modelli vengono sempre simulati in più configurazioni iniziali così da poter osservare come varia il loro comportamento con la presenza di perturbazioni iniziali. Le quantità di ogni specie vengono perturbate di una certa percentuale indicata dall'utente. Ciò garantisce che quantità positive non possano mai diventare negative facendoci ottenere configurazioni errate.

Memorizzazione dei risultati

Una simulazione restituisce come risultato un array Python che mostra come variano le quantità/concentrazioni delle specie nel tempo [1]. Questi dati vengono salvati in due file con estensione `.txt`. Il primo file li conterrà tutti e con una precisione maggiore rispetto al secondo, il quale ne conterrà solo una parte, considerando intervalli temporali maggiori e una minore precisione (vengono considerate meno cifre decimali). I risultati memorizzati in questi file sono strutturati in colonne: la prima rappresenta il tempo, mentre le successive rappresentano le specie presenti nella rete.

Consideriamo, ad esempio, la CRN definita dalla Tabella 2.2, i file di testo generati dalle sue simulazioni avranno la seguente struttura:

```
# ['time', '[species_0]', '[species_3]', '[species_1]', '[species_4]', '[species_2]', '[species_5]']
0.0000000000 1.0236670000 3.2036920000 2.1542310000 8.2321920000 6.2717930000 9.3120210000
0.2000000000 1.0236673661 3.2037010599 2.1542349488 8.2321838420 6.2717893664 9.3120193806
0.4000000000 1.0236676566 3.2037051855 2.1542368800 8.2321793211 6.2717879565 9.3120186444
0.6000000000 1.0236678352 3.2037066061 2.1542378548 8.2321769026 6.2717876667 9.3120182936
0.8000000000 1.0236679366 3.2037062295 2.1542381983 8.2321758937 6.2717881171 9.3120181960
1.0000000000 1.0236679821 3.2037052569 2.1542383945 8.2321754116 6.2717887700 9.3120181408
1.2000000000 1.0236680094 3.2037041561 2.1542385662 8.2321750706 6.2717894591 9.3120180890
1.4000000000 1.0236680230 3.2037030239 2.1542387465 8.2321747958 6.2717901449 9.3120180276
1.6000000000 1.0236680499 3.2037022224 2.1542390104 8.2321743413 6.2717907021 9.3120179357
1.8000000000 1.0236680768 3.2037014208 2.1542392743 8.2321738869 6.2717912594 9.3120178438
2.0000000000 1.0236681037 3.2037006192 2.1542395382 8.2321734324 6.2717918166 9.3120177519
```

dove, ad esempio, le specie 'species_1' e 'species_2' corrispondono rispettivamente alla quarta e sesta colonna e al tempo 1 le specie sono presenti con quantità/concentrazioni indicate nella settima riga del file.

I nomi dei file creati sono significativi e indicano la specie la cui quantità è stata modifica e la percentuale di incremento o decremento. Ad esempio, considerando la struttura di file vista in precedenza, se la specie modificata è la prima ed è stata diminuita del 10%, i nomi dei file contenenti i risultati della nuova configurazione saranno così definiti:

`Results_species_0_minus_10_p`
`Small_Results_species_0_minus_10_p`

rispettivamente per il file contenente tutti i dati generati e per il file che, invece, ne contiene solo una parte.

Simulazioni

Tra le funzioni definite nel modulo, ne compare una per simulare i modelli per cui è noto (grazie ai teoremi di Feinberg) che raggiungono lo steady state: `simulationWithSteadyState`. L'idea è quella di fermare le simulazioni quando lo stato stazionario viene raggiunto. libRoadRunner permette di calcolare le quantità di molecole possedute dalle specie allo stato stazionario grazie alla funzione seguente [1]:

`getSteadyStateValues()`

Questa può, però, generare eccezioni e quindi, per alcuni modelli potrebbe non restituire i valori cercati. Non verrà usata per distinguere modelli che presentano oscillazioni da quelli che raggiungono, invece, lo steady state, in quanto può ritornare valori anche per modelli che rientrano nella prima categoria. È, inoltre, possibile che alcune delle quantità restituite siano negative. In questo caso i valori trovati vengono considerati non attendibili. Per questo motivo, la simulazione tiene conto delle due situazioni: quella in cui si hanno quantità positive per tutte le specie e quella in cui, invece, queste informazioni non sono disponibili. La simulazione di una configurazione è racchiusa in una funzione, `simulationSteps`, che determina automaticamente il comportamento più adatto in base ai valori in input. Nel primo caso, viene eseguita una simulazione per passi finché non si ottengono quantità pari a quelle ottenute con la funzione sopra nominata. Nel secondo caso, la simulazione viene ancora eseguita per passi, ma essa termina quando, osservando i dati, sembra che lo steady state sia stato raggiunto, ovvero quando, confrontando tra loro tutte le righe ottenute dal passo di simulazione (non considerando la colonna associata al tempo), le quantità qui presenti sono, per ogni specie, uguali in ogni istante, considerando una precisione alla quinta cifra decimale. Se alcune delle quantità presenti risultano negative la simulazione viene fermata.

In Figura 4.15 è possibile osservare i controlli eseguiti per verificare se lo steady state è stato raggiunto, dove `values` è il vettore contenente il risultato della funzione `getSteadyStateValues()`. Si possono notare i due differenti comportamenti in base alla disponibilità di questa lista.

Vengono simulate tutte le configurazioni iniziali ottenute perturbando le quantità in cui sono presenti le specie, chiamando la funzione appena descritta per ogni configurazione. Per modificare le quantità delle specie viene utilizzata la funzione di libRoadRunner:

`setFloatingSpeciesAmounts`

```
368         if values!=[]:
369             results=sim[points-1,1:]
370             for i in range(len(results)):
371                 results[i]="{0:.5f}".format(results[i])
372         else:
373 
374             first=sim[0,1:].round(decimals=5)
375             last=sim[points-1,1:].round(decimals=5)
376 
377             if (first==last).all():
378                 stop=True
379                 for i in range(points-1):
380                     #se è presente una riga diversa dalle altre
381                     if not ((sim[i,1:].round(decimals=5)==last).all()):
382                         stop=False
383 
384             i=0
385             while(i<len(values) and last[i]>=0):
386                 i=i+1
387             if i<len(values):
388                 #valori negativi durante la simulazione
389                 break
```

Figura 4.15: Controllo del raggiungimento dello stato stazionario.

la quale permette di modificare le quantità di tutte le specie, prendendo in input la lista con i nuovi valori, oppure una sola, prendendo in input l'indice della specie e la nuova quantità da assegnarle [1].

Il salvataggio dei risultati delle simulazioni avviene all'interno della stessa funzione dove avviene la simulazione stessa.

Un altro metodo definito per le simulazioni di un modello utilizza un tempo fisso indicato al momento della chiamata, non ricorrendo, quindi, a una simulazione per passi. Anche in questo caso vengono generati i due file.

In Figura 4.16 viene mostrata l'esecuzione di una simulazione e il suo salvataggio sui file, indicati con `allResults` e `smallResults` rispettivamente per quello che contiene tutti i dati generati e quello che considera intervalli temporali maggiori specificati da `_interval`. Quindi, si osserva che la CRN viene simulata dal tempo 0 al tempo indicato (`simulationTime`) generando un numero di punti pari a `points`. Successivamente i risultati ottenuti vengono salvati sui file memorizzando in essi anche l'header, come definito in precedenza, ottenuto grazie alla funzione `timeCourseSelections` della libreria libRoadRunner.

```
291             try:
292                 sim=rr.simulate(0,simulationTime, points)
293                 np.savetxt(allResults, sim, fmt='%.8f',\
294                     header=str(rr.timeCourseSelections))
295 
296                 np.savetxt(smallResults, [sim[0,:]], fmt='%.5f',\
297                     header=str(rr.timeCourseSelections))
298                 k=_interval
299                 while k<len(sim):
300                     np.savetxt(smallResults, [sim[k,:]], fmt='%.5f')
301                     k=k+_interval
302             except:
303                 print("Error")
```

Figura 4.16: Simulazione utilizzando il tempo indicato.

Infine, in questo modulo sono state definite altre due funzioni, le quali simulano cercando di capire il comportamento del sistema studiato. La prima si occupa della simulazione in batteria iterando sui file nella directory indicata e chiamando la seconda per i file che presentano estensione `.xml`. È possibile osservarne il codice in Figura 4.17.

La seconda funzione, `simulateModel`, studia un singolo modello per volta cercando di capire se esso oscilla o se raggiunge lo steady state. Al suo interno, vengono create le cartelle che conterranno i risultati, i quali vengono suddivisi in base al comportamento presentato durante l'analisi della configurazione iniziale definita nel modello.

Questo viene caricato da libRoadRunner e studiato attraverso il procedimento illustrato in Figura 4.18.

Qui si può osservare che la simulazione viene eseguita per passi finché non viene intuito l'andamento del sistema o non viene raggiunto il tempo massimo di simulazione indicato dalla variabile `simulationTime`. Il comportamento del modello può essere individuato in tempo minore rispetto a quello indicato in `simulationTime`. Per ogni passo di simulazione, i dati generati vengono salvati in due file (strutturati come descritto in precedenza) e successivamente vengono

```
13  def simulateAllModels(rr, dirToSimulation, toPath, dirReject,\
14       simulationTime, time, percent, points=_points):
15 
16      files=os.listdir(dirToSimulation)
17      for file in files:
18          if file.endswith('.xml'):
19              print(file)
20              simulateModel(rr, dirToSimulation, file, toPath,\
21                   dirReject, simulationTime, time, percent, points)
```

Figura 4.17: Simulazione in batteria dei modelli.

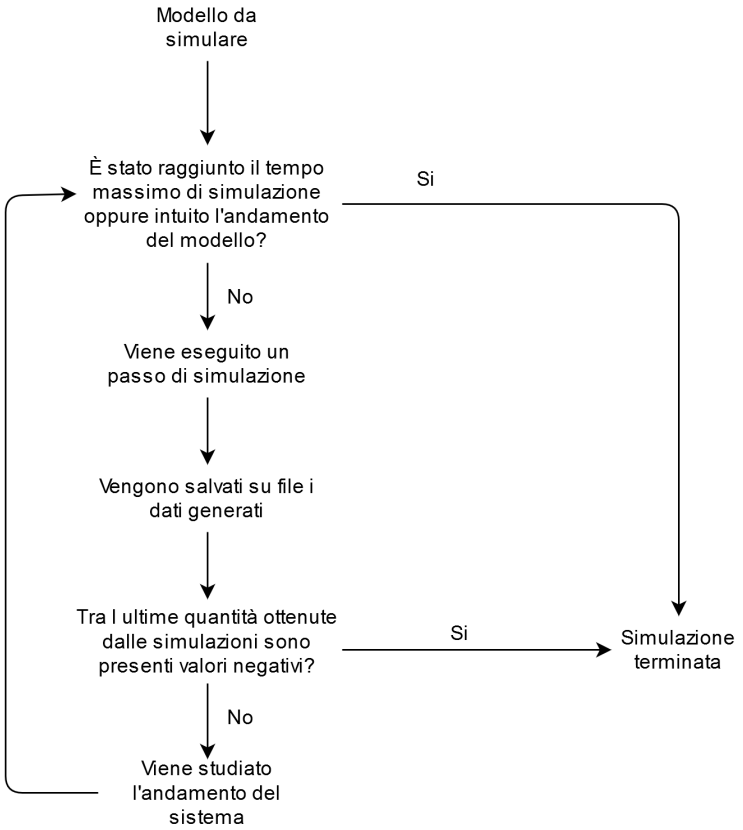


Figura 4.18: Simulazione per passi che analizza l'andamento di una rete di reazioni chimiche.

analizzati. Nel caso in cui al termine del passo di simulazione sono presenti specie con quantità negative allora la simulazione termina anche se non è stato raggiunto il tempo massimo specificato. In questo caso il comportamento del modello viene indicato come non noto. Se, invece, le quantità sono tutte positive viene studiato l'andamento del sistema. Per fare ciò viene eseguito un primo confronto tra le quantità presenti all'inizio del passo di simulazione e quelle che compaiono alla fine. Se queste non coincidono, viene cercata quest'ultima configurazione nel passo di simulazione precedente. Se viene trovata allora viene indicato che il modello presenta oscillazioni, in quanto la stessa configurazione è stata trovata due volte, ma tra i due momenti considerati le quantità sono cambiate. In Figura 4.19 è possibile osservare due passi di simulazione di un modello che oscilla, il quale è composto da due specie. Supponiamo di studiare il Passo *i+1* in figura con il procedimento sopra descritto. Per prima cosa vengono confrontate, per il passo considerato, le configurazioni iniziale e finale. Dato che non coincidono, si procede cercando quest'ultima nel passo di simulazione

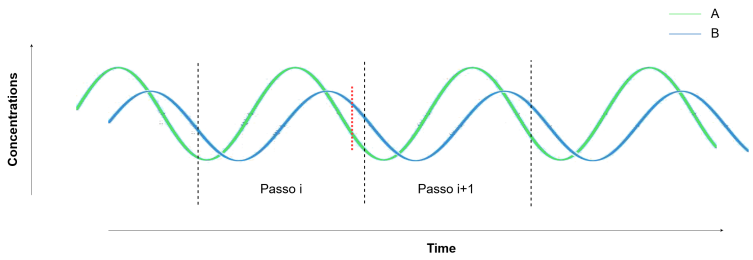


Figura 4.19: Simulazione per passi di un modello che oscilla.

precedente, i, in cui viene trovata (in figura rappresentata in rosso). Viene indicato che il modello presenta oscillazioni, in quanto la stessa configurazione è stata trovata in intervalli temporali diversi tra i quali le concentrazioni delle specie, e con esse le quantità, sono, però, variate.

Viene fatto un ragionamento analogo nel caso in cui la prima e l'ultima configurazione del passo di simulazione siano uguali, ma quelle che compaiono tra le due siano diverse da esse. Ne caso in cui, invece, tutte le configurazioni generate dal passo di simulazione siano uguali, viene considerato raggiunto lo stato stazionario. Tutti questi confronti vengono eseguiti considerando una certa tolleranza.

Se il tempo massimo viene raggiunto la simulazione termina anche se l'andamento del sistema non è stato intuito.

Dopo che la simulazione appena descritta è terminata si procede alle simulazioni che presentano quantità iniziali perturbate. Nel caso in cui l'analisi precedente abbia evidenziato oscillazioni o non abbia intuito il comportamento del modello considerato, le simulazioni con le nuove configurazioni non avvengono per passi e utilizzano il tempo fittizio preso in input dalla funzione.

```
110             if not (first==last).all():···
118             else:
119                 #ricerca di una configurazione intermedia diversa
120                 #dalla prima e l'ultima
121                 for j in range(points-1):
122                     if not (sim[j,1:].round(decimals=5)==last).all():
123                         stop=True
124                         toPath=toPath+"\\Oscillation"
125                         print("Oscillation")
126                         break
127                 if not stop:
128                     #se la configurazione intermedia non è stata trovata
129                     stop=True
130                     steadyState=True
131                     toPath=toPath+"\\Steady_State"
132                     print("Steady state")
```

Figura 4.20: Analisi dell'andamento del sistema.

Se, invece, la precedente simulazione era terminata perché era stato raggiunto lo steady state, quelle successive vengono eseguite utilizzando la funzione `simulationSteps` illustrata in precedenza, dove in input viene indicato di non avere informazioni circa le quantità presentate dalle specie allo stato stazionario.

In Figura 4.20 è possibile osservare l'analisi dell'andamento del sistema nel caso in cui la prima configurazione (`first`) e l'ultima (`last`) siano uguali. Si può osservare l'iterazione sui risultati intermedi, con l'esclusione della prima colonna corrispondente al tempo, e l'aggiornamento del percorso dove memorizzare i risultati.

4.5 Modulo 5: PetriNets

Questo modulo si occupa della creazione della Petri net di uno o più modelli. Per fare ciò sono stati definiti due metodi: `createPetriNets` e `createOnePetriNet`. Il primo permette di creare le Petri nets di tutti i modelli presenti ad un dato percorso, richiamando, per ognuno di essi, il secondo. In Figura 4.21 si può osservare l'implementazione di questa funzione. Essa prende in input i seguenti parametri:

- `rr`, che rappresenta il riferimento a libRoadRunner;
- `path`, che corrisponde al percorso dove cercare i modelli;
- `pathResult`, che indica il percorso dove salvare i risultati;
- `nTest` e `valueIncrease`, che sono parametri usati per analizzare il comportamento di alcune specie. L'uso di questi ultimi verrà meglio illustrato in seguito.

Inizialmente, `createOnePetriNet` crea la cartella dove memorizzare i risultati utilizzando la funzione del modulo definito nel Paragrafo 4.1. Successivamente carica il modello sia con libRoadRunner che con libSBML e inizia a costruire la Petri net, a cui viene dato il nome del file analizzato. I primi elementi rappresentanti sono i posti, le specie, a cui, come etichetta, viene assegnato il loro id. Questo può essere ottenuto mediante l'uso delle seguenti funzioni della libreria libSBML:

```
model.getNumSpecies()
model.getSpecies(i).getId()
```

```
 8  def createPetriNets(rr, path, pathResult, nTest, valueIncrease):
 9      files=os.listdir(path)
10      for file in files:
11          if file.endswith('.xml'):
12              print(file)
13              #creazione di una Petri net
14              createOnePetriNet(rr, path, file, pathResult, nTest, valueIncrease)
```

Figura 4.21: Creazione delle Petri nets di una batteria di modelli.

dove la prima permette di ottenere il numero delle specie del sistema e la seconda di leggere l'id associato all'i-esima specie. Possiamo quindi ottenere tutti gli id utilizzando un ciclo `for` per scorrere tutte le specie. In seguito, vengono analizzate le reazioni, una alla volta e ne viene letto l'id (anche in questo caso utilizzato come etichetta) con una funzione simile alla precedente. Questi nodi che corrispondono alle transizioni vengono definiti con una forma rettangolare (`[shape=box]`), mentre i posti presentano forma circolare (`[shape=circle]`). Vengono rappresentate anche le reazioni inverse, per le quali viene definito un ulteriore nodo che presenta un'etichetta pari all'id della reazione concatenato alla stringa "_Reverse".

Ad esempio se una certa reazione invertibile ha come identificativo "R1", l'etichetta associata all'inversa sarà "R1_Reverse".

Per ogni reazione, comprese le inverse, vengono poi rappresentati gli archi tra posti e transizioni, cioè tra specie e reazioni. Al momento dell'aggiunta dell'arco, ad esso si associa l'etichetta con il coefficiente stochiometrico della specie. Tutte queste informazioni sono ricavabili tramite la libreria libSBML. Devono essere rappresentate anche le specie che si comportano come promotori o inibitori per la reazione, distinguendo tra i due casi. Per questo motivo il loro comportamento viene analizzato e, in base ai risultati dell'analisi, viene scelto l'arco appropriato.

Nella Figura 4.22 è mostrato il codice per la creazione degli archi di una Petri nets, dove `functionality` è la funzione per analizzare il comportamento del modificatore. Questa prende tra i suoi argomenti il modello analizzato `model`, l'indice del modificatore analizzato, `j`, l'indice della reazione, `i`, e i parametri `nTest` e `valueIncrease` nominati in precedenza.

Analizzando i reagenti (o i prodotti) il comando per ottenere l'id della specie è il seguente:

```
s.getSpecies()
```

dove `s` rappresenta il reagente (o il prodotto). Si può notare che durante l'analisi di reagenti, prodotti o modificatori, per ottenere l'id della specie non viene usata la funzione `getId()`, usata invece durante la creazione dei posti. Ciò è dovuto al fatto che le funzioni seguenti restituiscono oggetti di classi differenti:

- `model.getSpecies(i)` restituisce un oggetto della classe `Species`;
- `model.getReaction(i).getListOfReactants()` restituisce una lista di oggetti della classe `SimpleSpeciesReference`.

L'attributo `Species` di un oggetto `SimpleSpeciesReference` è obbligatorio e corrisponde all'id di un oggetto della classe `Species` [13].

Come detto in precedenza, devono essere rappresentate anche le specie che si comportano da modificatori e per questo motivo viene studiato come esse influenzano la velocità della reazione. Per prima cosa viene verificato che la specie appaia nella legge cinetica della reazione, ossia nella formula che indica come viene calcolata la velocità.

```
61          #per ottenere i reagenti della reazione
62          for reactant in model.getReaction(i).getListOfReactants():
63              graph=graph+str(reactant.getSpecies())+" -> "+\
64                  str(model.getReaction(i).getId())+\
65                  " [arrowhead=vee, label="+str(reactant.getStoichiometry())+"];\n"
66
67          #per ottenere i prodotti della reazione
68          for product in model.getListOfReactions().get(i).getListOfProducts():
69              graph=graph+str(model.getReaction(i).getId())+" -> "+\
70                  str(product.getSpecies())+\
71                  " [arrowhead=vee, label="+str(product.getStoichiometry())+"];\n"
72
73          #per ottenere i modificatori della reazione
74          for j in range(model.getReaction(i).getNumModifiers()):
75              response=functionality(rr, model, j, i, nTest, valueIncrease)
76              #se il modificatore si comporta da promotore
77              if response==1:
78                  graph=graph+str(model.getReaction(i).getModifier(j).getSpecies())+\
79                      " -> "+str(model.getReaction(i).getId())+" [arrowhead=dot];\n"
80              #se il modificatore si comporta da inibitore
81              elif response==0:
82                  graph=graph+str(model.getReaction(i).getModifier(j).getSpecies())+\
83                      " -> "+str(model.getReaction(i).getId())+" [arrowhead=tee];\n"
```

Figura 4.22: Implementazione della creazione degli archi.

Per fare ciò viene letto l'elemento `KineticLaw`, contenuto nell'oggetto rappresentante la reazione e ne vengono analizzati i nodi, ossia gli elementi. Se tra questi la specie considerata non viene trovata allora vuol dire che non ha nessuna influenza sulla velocità della reazione. In questo caso la funzione termina restituendo un apposito codice. Se, invece, la specie appare nella formula si tenta di capirne il ruolo analizzando come varia la velocità al variare della quantità di molecole. Questa quantità viene incrementata di una certa percentuale dipendente dalla variabile `valueIncrease`. Più quest'ultima aumenta, più l'incremento è maggiore. Vengono fatti un numero di incrementi pari al valore di `nTest` e per ognuno di essi, la velocità della reazione con la nuova quantità viene confrontata con la precedente. Nel caso in cui la velocità ottenuta risulti maggiore della precedente dopo ogni aumento, la specie viene considerata un promotore, altrimenti, se dopo ogni aumento risulta minore, si comporta come inibitore.

È possibile osservare l'implementazione del procedimento appena descritto nella Figura 4.23. Per modificare la quantità della specie e poter leggere la velocità della reazione analizzata sono stati usati i seguenti metodi della libreria libRoadRunner:

- `rr.model.getReactionRates()[k]` per ottenere la velocità della k-esima reazione;
- `rr.model.setFloatingSpeciesAmounts([i],[j])` per assegnare all' i-esima specie la quantità j.

```
208         #indice delle specie da esaminare
209         index=rr.model.getFloatingSpeciesIds().index(\
210             model.getReaction(reaction).getModifier(specie).getSpecies())
211
212         #velocità iniziale posseduta dalla reazione
213         originalRate=rr.model.getReactionRates()[reaction]
214         #variabili utilizzate per studiare il comportamento della specie
215         countProm=0
216         countInhib=0
217         #vengono eseguiti più test
218         for i in range(1, nTest+1):
219             #icremento della quantità della specie da analizzare
220             rr.model.setFloatingSpeciesAmounts([index],[quantities[index]+\
221                 quantities[index]*(i*valueIncrease)/100])
222             #se la velocità aumenta
223             if rr.model.getReactionRates()[reaction]>originalRate:
224                 countProm=countProm+1
225             #se la velocità diminuisce
226             elif rr.model.getReactionRates()[reaction]<originalRate:
227                 countInhib=countInhib+1
228             #aggiornamento della velocità per il prossimo confronto
229             originalRate=rr.model.getReactionRates()[reaction]
230         #se la velocità è diminuita ad ogni iterazione la specie
231         #si comporta da inibitore
232         if countInhib==nTest:
233             return 0
234         #se la velocità è aumentata ad ogni iterazione la specie
235         #si comporta da promotore
236         elif countProm==nTest:
237             return 1
```

Figura 4.23: Analisi del comportamento di un modificatore.

Nel caso in cui il comportamento della specie continui a non essere noto, viene tentato un incremento maggiore dipendente sempre dalla variabile `valueIncrease`. Se anche in questo modo rimane sconosciuto, si modificano le condizioni iniziali, ovvero vengono cambiate anche le quantità delle altre specie. Successivamente, per ognuna di queste configurazioni viene applicato lo stesso procedimento descritto in precedenza. Se dopo aver analizzato un numero di configurazioni pari a `nTest` il ruolo della specie continua ad essere non noto la funzione termina restituendo un apposito codice. Gli archi tra un modificatore e una reazione non presentano etichette.

La Petri net viene salvata su un file `.gv` usando i seguenti comandi per la gestione di file, offerti dal linguaggio Python [16]:

```
f=open(path+nameFile+".gv", "w")
```

per aprire il file nameFile.gv che si trova al percorso dato. Se questo non esiste allora viene creato. La stringa ‘‘w’’ indica che il file viene aperto in scrittura,

```
f.write(petriNet)
```

```
f.close()
```

rispettivamente per scrivere la stringa indicata nel file e per chiuderlo. Nel caso in cui si verifichi un errore la funzione per creare la Petri net ritorna un apposito codice.

4.6 Modulo 6: Main

Il flusso del programma, illustrato brevemente all'inizio del Capitolo 4, è definito in questo modulo, il quale è composto da due metodi. Il primo presenta la seguente segnatura:

```
main(interval=-1, maxInterval=20, dirToExamine="models", func=0)
```

dove `interval` indica l'intervallo di perturbazione delle quantità delle specie e `maxInterval` indica la massima perturbazione possibile. Ad esempio, se `interval=5` e `maxInterval=20` allora l'incremento avviene ad intervalli di 5, cioè del 5%, del 10%, fino ad arrivare ad un massimo del 20%. `dirToExamine` corrisponde alla cartella contenente i file da analizzare e `func` permette di scegliere se eseguire l'intero programma oppure solo una sua parte. È quindi possibile eseguire una tra le seguenti funzionalità indicando l'apposito codice numerico:

- l'esecuzione dell'intero programma, ovvero la selezione dei modelli, la loro analisi con i teoremi di Feinberg, la simulazione di tutti i modelli di interesse e la creazione delle Petri nets. Questa scelta è rappresentata con il numero '0';
- la selezione dei modelli di interesse secondo i criteri illustrati nel paragrafo 4.2, rappresentata con '1';
- l'analisi dei modelli con i teoremi di Feinberg, quando possibile, rappresentata con '2';
- la simulazione dei modelli indicati cercando di intuirne l'andamento grazie alla funzione definita nel Paragrafo 4.4, rappresentata con il numero '3';
- la creazione delle Petri nets dei modelli indicati, rappresentata con il numero '4'.

I parametri in input sono tutti opzionali e, nel caso in cui al momento della chiamata del metodo non vengano indicati dei valori, vengono usati i valori di default indicati nella segnatura del metodo. Il parametro `interval` è necessario nel caso in cui siano richieste simulazioni. Nel caso in cui l'utente scelga di eseguire una funzionalità che prevede simulazioni, ma non indichi nessun valore al momento della chiamata, verrà chiesto all'inizio dell'esecuzione di inserire l'informazione necessaria da tastiera. Per questa richiesta è stata utilizzata la funzione definita nel Paragrafo 4.1. Nel caso in cui il valore indicato non sia

```
29      for file in files:
30          if file.endswith('.xml'):
31              print(file)
32              rr.clearModel()
33
34              try:
35                  rr.load(dirToExamine+'\\'+file)
36              except:
37                  shutil.move(dirToExamine+'\\'+file, dirNotAnalyze)
38                  print('Not_Analyze')
39                  continue
40
41              #controllo dell'applicabilità dei teoremi
42              if Simulation.toSimulate(rr)==1:
43                  #i teoremi non sono applicabili
44                  shutil.move( dirToExamine+'\\'+file, dirToSimulation)
45                  continue
46
47              #analisi del modello con i teoremi di Feinberg
48              results=Deficiency_Calculation.deficiency_calculation(rr)
49              if results[1]==1:
50                  print("Teoremi di Feinberg non applicabili al modello")
51                  shutil.move(dirToExamine+'\\'+file, dirToSimulation)
52              elif results[0]==0 or results[0]==1:
53                  print("Applicati i Teoremi di Feinberg: Raggiunge lo steady state")
54                  shutil.move(dirToExamine+'\\'+file, dirTheorems+"\\SteadyState")
55              else:
56                  print("Teoremi applicabili: comportamento non noto")
57                  shutil.move(dirToExamine+'\\'+file, dirTheorems+"\\not_known")
```

Figura 4.24: Applicazione dei teoremi di Feinberg a una batteria di modelli.

un valore valido il programma termina restituendo un errore. Tutti gli altri parametri utilizzati sono fissati.

Analizziamo ora il funzionamento dell'intera esecuzione del programma. Dato che i modelli verranno simulati, all'inizio è richiesto un valore valido per la variabile `interval`. Vengono calcolati gli intervalli, allontanandosi dalle quantità originali in cui sono presenti le specie fino a raggiungere il massimo indicato dalla variabile `maxInterval`. Successivamente viene chiamato il metodo `select` definito nel Paragrafo 4.2 e il secondo metodo definito all'interno di questo modulo `Main` che applica i teoremi di Feinberg ai modelli che non vengono esclusi dalla selezione precedente. Questa funzione prende in input le cartelle da analizzare e quelle che conterranno i modelli, suddivisi in base al criterio con cui devono essere studiati. Se queste non esistono vengono create. Ogni modello presente nella cartella indicata viene esaminato utilizzando la libreria libRoadRunner (leggendolo mediante il metodo `load`) e viene successivamente studiato con la prima funzione descritta nel Paragrafo 4.4 che stabilisce se è possibile studiare il modello con i teoremi di Feinberg. In caso di risposta positiva viene calcolata la deficiency della CRN e verificata la proprietà di reversibility o weakly reversibility utilizzando le funzioni presenti nel modulo illustrato nel Paragrafo 4.3. Nel caso in cui la rete non possieda queste ultime caratteristiche

allora non è possibile studiare il modello con i teoremi e il modello viene spostato nella directory apposita. Nel caso in cui, invece, una delle proprietà valga, viene controllata la deficiency del sistema. Se essa è pari a 1 o 0 allora la rete raggiunge lo stato stazionario, altrimenti non si può fare nessuna assunzione sul comportamento del modello. Questi modelli vengono spostati nelle cartelle apposite. La directory per i modelli con comportamento non noto a cui sono stati applicati i teoremi è diversa da quella in cui sono presenti i modelli a cui, invece, non sono applicabili perché si è preferito distinguere i due casi per poter fare analisi sui risultati ottenuti. In Figura 4.24 è possibile osservare una parziale implementazione della funzione appena descritta.

Dopo aver stabilito quali sono i modelli che raggiungono lo steady state e quali hanno un comportamento non noto, i primi vengono simulati con la funzione definita nel Paragrafo 4.4 e gli altri, invece, mediante il metodo che analizza l'andamento, illustrato anch'esso nel Paragrafo 4.4. Infine vengono create le Petri nets di tutti i modelli analizzati richiamando la funzione `createPetriNets` descritta nel Paragrafo 4.5.

Tutti i parametri necessari alle funzioni chiamate sono definiti nella funzione `main`, come, ad esempio, il tempo fissato per la simulazione dei modelli che presentano oscillazioni o il numero di test usato per l'analisi dei modificatori durante la creazione delle Petri nets. La scelta di questi valori verrà meglio motivata nel Capitolo 5.

4.7 Creazione del package

Dopo aver delineato i moduli e le funzioni al loro interno, viene creato un package pronto per l'installazione e l'uso. Per crearlo è necessario definire un file `__init__.py` all'interno della cartella da trasformare in un package [12]. Il file `__init__` usato è vuoto. Inoltre, al package sono associate una licenza e una descrizione memorizzate rispettivamente nei file `LICENSE.txt` e `README.md` [3]. La licenza scelta è quella di Apache, in particolare la versione 2.0 [2]. È una licenza permissiva che può essere usata per usi sia commerciali che privati e permette di apportare modifiche e creare software derivati [6]. Come mostrato in Figura 4.25, questa non offre nessuna garanzia agli utilizzatori del software realizzato [2], il quale è offerto "così com'è" ("AS IS").

Dopo aver definito questi due file, per la costruzione del package installabile viene creato lo script `setup.py`, che utilizza la libreria `setuptools`, in particolare i moduli per il setup e la ricerca dei package. In esso vengono indicate le seguenti informazioni [3]:

- `name` che definisce il nome del pacchetto che si andrà a creare per la distribuzione;
- `version` che indica la versione del software;
- `url` rappresenta l'URL dell'homepage dove trovare il package;

```
Copyright 2020 [Università di Pisa]

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

Figura 4.25: Licenza Apache 2.0.

- `license` che indica la licenza del software, definita nel file nominato in precedeza;
- `author` e `author_email` sono usati per identificare lo sviluppatore del package grazie a nome ed e-mail;
- `description` e `long_description` rappresentano rispettivamente una breve descrizione del progetto e una descrizione più approfondita. Quest'ultima è stata memorizzata nel `README` del pacchetto;
- `packages` identifica i package che devono essere inclusi in quello che verrà distribuito. In questo caso ne è presente solo uno, ma per individuarlo usiamo la funzione `find_packages`. Questa determina automaticamente quali package devono essere inclusi riconoscendoli grazie al file `__init__.py`. La loro ricerca inizia dalla cartella contenente il file di setup cercando anche nelle sottocartelle;
- `install_requires` sono i vincoli che devono essere soddisfatti per poter installare il pacchetto.

In Figura 4.26 è possibile osservare il file di setup definito per la costruzione del package. Si può, ad esempio, notare l'uso dei file sopra delineati oppure la definizione dei requisiti richiesti per l'installazione del pacchetto che, in questo caso, corrispondono all'installazione della libreria libRoadRunner e della libreria libSBML, rispettivamente con una versione pari o superiore alla 1.5.4 e alla 5.18.0 e delle librerie numpy e pathlib.

Adesso si hanno a disposizione tutti gli elementi per la costruzione del package pronto per l'installazione. Per crearlo si utilizza il seguente comando [12]:

```
python setup.py sdist
```

la cui esecuzione è mostrata in Figura 4.27.

```python
from setuptools import setup, find_packages

setup(name='SBML_batch',
      version='0.1',
      url='http://groups.di.unipi.it/msvbio/software/SBML_batch_simulator.html',
      license='LICENSE.txt', #Apache License 2.0
      author='Mariagiovanna Rotundo',
      author_email='mariagiovannarotundo@gmail.com',
      description='A python package for batch simulation of models ',
      packages=find_packages(),
      long_description=open('README.md').read(),
      install_requires=['libroadrunner>=1.5.4', 'libsbml>=5.18.0', 'numpy', 'pathlib']
      )
```

Figura 4.26: File di setup per la costruzione del package.

In questo modo viene creato un archivio compresso che contiene tutte le informazioni necessarie per essere installato e successivamente utilizzato. Ora il package è pronto per essere distribuito.

```
running sdist
running egg_info
creating SBML_batch.egg-info
writing SBML_batch.egg-info\PKG-INFO
writing dependency_links to SBML_batch.egg-info\dependency_links.txt
writing requirements to SBML_batch.egg-info\requires.txt
writing top-level names to SBML_batch.egg-info\top_level.txt
writing manifest file 'SBML_batch.egg-info\SOURCES.txt'
reading manifest file 'SBML_batch.egg-info\SOURCES.txt'
writing manifest file 'SBML_batch.egg-info\SOURCES.txt'
running check
creating SBML_batch-0.1
creating SBML_batch-0.1\SBML_batch
creating SBML_batch-0.1\SBML_batch.egg-info
copying files to SBML_batch-0.1...
copying README.md -> SBML_batch-0.1
copying setup.py -> SBML_batch-0.1
copying SBML_batch\BaseFunctions.py -> SBML_batch-0.1\SBML_batch
copying SBML_batch\Deficiency_Calculation.py -> SBML_batch-0.1\SBML_batch
copying SBML_batch\Main.py -> SBML_batch-0.1\SBML_batch
copying SBML_batch\PetriNets.py -> SBML_batch-0.1\SBML_batch
copying SBML_batch\SelectFiles.py -> SBML_batch-0.1\SBML_batch
copying SBML_batch\Simulation.py -> SBML_batch-0.1\SBML_batch
copying SBML_batch\__init__.py -> SBML_batch-0.1\SBML_batch
copying SBML_batch.egg-info\PKG-INFO -> SBML_batch-0.1\SBML_batch.egg-info
copying SBML_batch.egg-info\SOURCES.txt -> SBML_batch-0.1\SBML_batch.egg-info
copying SBML_batch.egg-info\dependency_links.txt -> SBML_batch-0.1\SBML_batch.egg-info
copying SBML_batch.egg-info\requires.txt -> SBML_batch-0.1\SBML_batch.egg-info
copying SBML_batch.egg-info\top_level.txt -> SBML_batch-0.1\SBML_batch.egg-info
Writing SBML_batch-0.1\setup.cfg
creating dist
Creating tar archive
removing 'SBML_batch-0.1' (and everything under it)
```

Figura 4.27: Esecuzione del comando '`python setup.py sdist`'.

Capitolo 5

Test

Nel modulo `Main` sono stati dichiarati i parametri richiesti dalle funzioni degli altri moduli che vengono chiamate. Alcuni di essi sono il risultato dello studio dei modelli considerati. Sono stati scaricati da `BioModels` 938 modelli, 709 dei quali non sono di nostro interesse a causa dei seguenti motivi, considerati nell'ordine indicato:

1. 7 di essi generano errori a tempo di caricamento da parte della libreria libRoadRunner;
2. in 552 compaiono regole;
3. in 42 dei rimanenti compaiono eventi;
4. 3 sono qualitativi;
5. 105 vengono letti in modo errato da libRoadRunner.

Nei modelli che presentano eventi non vengono contati quelli in cui sono definite anche delle regole, in quanto il controllo della presenza di queste ultime avviene prima, controllando la presenza di eventi solo nei modelli rimanenti.

Quindi, in totale vengono simulati 229 modelli. A 7 di questi è possibile applicare i teoremi di Feinberg, ma solo 5 raggiungono lo steady state. Non è possibile studiare il comportamento degli altri 2 in quanto generano errori a tempo di simulazione presentando valori negativi. Le analisi per la scelta dei parametri sono state svolte su questi 229 modelli.

Scelta dei parametri

I parametri definiti necessari alle funzioni chiamate dalla funzione `main`, descritta nel Paragrafo 4.6, riguardano:

- le directory utilizzate per distinguere i modelli che presentano comportamenti differenti e per memorizzare i risultati generati dal programma;

- i tempi di simulazione;
- il numero di test e gli incrementi delle quantità delle specie della rete per studiare i modificatori delle reazioni al momento della creazione delle Petri nets.

Le directory definite sono molte, quindi si è preferito usare nomi specifici significativi e non richiederli, invece, in input all'utente. Alcune directory utilizzate sono quella per memorizzare le Petri nets e quella per memorizzare i risultati delle simulazioni dei modelli a cui non sono stati applicati i teoremi di Feinberg, chiamate rispettivamente `PetriNets` e `Results_Simulations`.

Riguardo i tempi di simulazione, per scegliere il più adatto per la simulazione per passi, sono stati testati diversi tempi massimi per cercare di capire l'andamento di un modello, non considerando quelli a cui non è stato possibile applicare i teoremi di Feinberg.

Nella Tabella 5.1 sono indicate le unità di tempo considerate e la percentuale di modelli di cui si è intuito l'andamento:

Unità di tempo	Percentuale
2500	59,0%
5000	69,8%
7500	73,0%
10000	74,3%
12500	77,0%
15000	77,5%
17500	77,5%
20000	77,9%
40000	79,7%

Tabella 5.1: Unità di tempo utilizzate per analizzare l'andamento dei modelli e rispettiva percentuale di modelli per cui il comportamento è stato identificato.

Considerando che alcuni modelli generano errori e valori non validi a tempo di simulazione, una percentuale di circa l'80% è stata ritenuta sufficiente e, per questo motivo, non sono stati testati tempi maggiori a quelli indicati in tabella. Si può osservare che considerando tempi minori, la percentuale cresce più rapidamente rispetto a quando vengono considerati tempi maggiori. Ad esempio tra 2500 e 5000 cresce del 10,8%, mentre tra 17500 e 20000 cresce dello 0,4%.

Per la scelta del tempo massimo è stato eseguito un ulteriore esame sui modelli per cui i teoremi di Feinberg ci garantiscono che viene raggiunto lo steady state. È stato calcolato il tempo medio per raggiungerlo, considerando le configurazioni iniziali, il quale è pari a circa 16500. Considerando questa informazione e quelle presenti in Tabella 5.1 è stato scelto di utilizzare come tempo 20000, in quanto presenta una percentuale sufficientemente alta, maggiore di quella presentata da 17500, ed è un tempo molto vicino a quest'ultimo e quindi a 16500. È stato scelto di non usare come tempo 40000 in quanto, nonostante il

notevole aumento di tempo, il numero di modelli per cui viene stabilito un andamento aumenta di una percentuale minima rispetto al tempo precedentemente considerato.

Per il tempo scelto è stato calcolato il tempo medio per individuare l'andamento non considerando i modelli per cui viene indicato un comportamento sconosciuto al termine della simulazione. Il tempo medio calcolato usando come tempo massimo 20000 è pari a 2000.

L'andamento intuito dalla simulazione può essere quello sbagliato, ad esempio per un modello che raggiunge lo steady state viene indicato che oscilla o viceversa. Questo può essere dovuto, ad esempio, al fatto che sono presenti oscillazioni che diminuiscono fino al raggiungimento dello steady state, oppure a oscillazioni su cifre decimali non considerate a causa della precisione utilizzata nel programma. Si può cercare di mitigare il primo problema indicando un tempo minimo prima del quale non viene fatta nessuna supposizione sull'andamento del sistema considerato, così da permettere ad esso di stabilizzarsi. In questo modo, alcuni modelli che all'inizio sembrano presentare oscillazioni, dopo questo tempo indicato sembrano presentare un andamento stabile con oscillazioni ridotte o quasi nulle prima di raggiungere lo steady state e ciò permette di classificarli in modo corretto. L'uso di un tempo minimo presenta quindi vantaggi e svantaggi. Permette di intuire il comportamento esatto di modelli altrimenti classificati in modo errato, come ad esempio il caso appena descritto, ma per alcuni modelli accade l'opposto. Un esempio è quello nominato in precedenza, dove le oscillazioni più sostenute si verificano all'inizio, ma successivamente il modello presenta oscillazioni talmente piccole che non vengono rilevate.

Nelle funzioni che cercano di capire l'andamento di un modello descritte nel Paragrafo 4.4 è possibile specificare un tempo minimo al momento della chiamata della funzione. Si potrebbe, ad esempio considerare il tempo medio calcolato come in precedenza per i tempi in tabella 5.1 utilizzandoli rispettivamente come minimo e massimo. È stato deciso di non utilizzare un tempo minimo quindi, nel metodo `main` questo è inizializzato a 0.

Quindi, eseguendo una simulazione per passi che presenta tempo massimo pari a 20000 e minimo pari a 0, i modelli che raggiungono lo stato stazionario sono 113, mentre quelli che presentano oscillazioni sono 60, sui 222 modelli totali a cui non è stato possibile applicare i teoremi di Feinberg.

Non è stato possibile fare alcun tipo di analisi sui modelli a cui è possibile applicare i teoremi, ma che non presentano una deficiency pari a 0 o 1 a causa degli errori che questi modelli hanno presentato a tempo di simulazione.

Riguardo i parametri relativi alle Petri nets è stato scelto un incremento con valore pari a 5 e un numero di Test pari a 10. Il primo è stato ritenuto adatto in quanto permette di analizzare il ruolo di un modificatore incrementando inizialmente la sua quantità ad intervalli del 5%. Questa percentuale è considerata adeguata in quanto gli incrementi non sono così piccoli da non influire sulla velocità delle reazione, ma le nuove quantità non si discostano troppo da quelle originali. Bisogna prestare attenzione alla scelta di questo valore in quanto, se questo provoca un incremento eccessivo delle quantità, è possibile che vengano violati vincoli riguardanti le specie presenti nel modello. Ad esempio, è stato

notato che in alcuni di essi, viene garantito che la quantità di alcune specie non superi mai un certo valore. In questo caso, un incremento troppo elevato, causato dalla scelta di un valore troppo alto, può portare ad fare un'analisi del modello dove i vincoli non sono rispettati. Se ciò accade, i risultati non possono essere considerati attendibili.

Per lo stesso motivo è stato scelto un numero di Test pari a 10, in quanto, dai test eseguiti con l'incremento nominato in precedenza, è stato notato che i vincoli presenti nei modelli (come quello sopra citato) vengono rispettati, dato che il numero di test non è tanto elevato da causare un incremento eccessivo delle quantità. Inoltre, il valore scelto viene ritenuto adatto perché permette di eseguire un numero significativo di test e di controlli sulle velocità per poter individuare il ruolo del modificatore.

Costruzione del dataset

Come detto in precedenza sono stati scaricati da `BioModels` 938 modelli i quali sono stati dati in input alla funzione `main` illustrata nel Capitolo 4.6. Solo 229 di questi sono stati selezionati per la simulazione a causa dei motivi sopra citati. Utilizzando la funzione definita nel Capitolo 4.4, 37 di essi sono stati riconosciuti come modelli studiati mediante la legge di azione di massa e sono stati successivamente analizzati con i teoremi definiti da Feinberg, usando il metodo illustrato nel Capitolo 4.3. In 7 di queste reti sono state verificate le proprietà di reversibility o weakly reversibility, ma solo 5 hanno presentato una deficiency pari a 0 o pari a 1. Dopo aver svolto le analisi sulla struttura delle reti, queste sono state simulate. Per i 222 modelli, per cui i teoremi non hanno garantito il raggiungimento dello stato stazionario, sono state eseguite simulazioni che tentano di intuirne il comportamento. Per 113 reti è stato individuato uno stato stazionario, mentre per 60 sono state notate oscillazioni. Infine, sono state create le Petri nets, una per ogni modello simulato. In Tabella 5.2 si possono osservare i tempi medi impiegati per eseguire queste operazioni sugli insiemi di modelli sopra citati. Il tempo medio di ogni operazione è stato ottenuto calcolando la media aritmetica dei tempi osservati in più esecuzioni della stessa funzionalità. Ad esempio, per calcolare il tempo medio necessario alla selezione dei modelli è stato utilizzato l'insieme composto dai 938 sopra menzionati. La selezione da questo insieme è stata eseguita più volte e, per ognuna di esse, è stato osservato il tempo utilizzato. Infine ne è stata calcolata la media. È stato eseguito un procedimento analogo per il calcolo dei tempi medi impiegati dalle altre operazioni, con la differenza che, per queste ultime, l'insieme usato è composto dai 229 modelli selezionati in precedenza.

Si può notare che l'operazione più costosa è la simulazione, eseguita in questo caso con una perturbazione del 20%. Per simulare tutti i modelli è stato impiegato un tempo medio di circa 6 ore e mezza. La seconda operazione più costosa riguarda la selezione dei modelli, il cui tempo medio è di circa 4 minuti, quindi molto inferiore rispetto a quello di simulazione.

Le operazioni che vengono eseguite più velocemente sono, invece, lo studio delle leggi cinetiche, e la conseguente applicazione dei teoremi di Feinberg e

Tipo di operazione	Tempo medio
Selezione dei modelli	04m 01s
Studio della legge cinetica e applicazione dei teoremi	43s
Simulazione (con intervalli del 20%)	06h 30m 46s
Creazione delle Petri nets	40s

Tabella 5.2

la creazione delle Petri nets. Queste richiedono tempi brevi e in particolare, mostrano tempi medi di circa 43 e 40 secondi.

Uno dei 938 modelli presi in input dal programma è quello con codice identificativo `BIOMD0000000184` [10] che presenta le specie e le reazioni mostrate in Tabella 5.3.

Specie	Id della reazione	Reazione	Modificatori
X	R1	$\emptyset \rightarrow X$	
Y	R2	$X \rightarrow \emptyset$	
Z	R3	$Y \rightarrow X$	Z
	R4	$X \rightarrow Y$	
	R5	$Y \rightarrow X$	
	R6	$\emptyset \rightarrow Z$	X
	R7	$Z \rightarrow \emptyset$	

Tabella 5.3: Specie e reazioni di una CRN. Ø indica che non sono presenti reagenti o prodotti.

Durante l'esecuzione del programma, questo modello viene selezionato per la simulazione in quanto è non qualitativo e non presenta né regole né eventi. Inoltre, viene letto in modo corretto dalla libreria libRoadRunner. Successivamente vengono studiate le leggi cinetiche delle reazioni e, data la presenza di modificatori, osservabili nelle reazioni R3 ed R6, alla rete non vengono applicati i teoremi di Feinberg. Per questo motivo si è ricorsa alla simulazione per passi che cerca di intuirne l'andamento e da essa sono state evidenziate oscillazioni. Il comportamento del sistema può essere osservato in Figura 5.1.

Questo modello presenta tre specie. Supponendo una simulazione con perturbazione del 20%, sono state considerate le seguenti configurazioni:

- la configurazione iniziale;
- per ognuna delle tre specie, le configurazioni ottenute incrementando e decrementando le quantità del 20%;

Quindi in totale sono state eseguite 7 simulazioni, ognuna delle quali è stata salvata su due file, come descritto nel Capitolo 4.4. Pertanto, per questo modello, considerando la perturbazione indicata in precedenza, sono stati generati 14 file contenenti i risultati delle simulazioni. Terminate le simulazioni, è stata creata

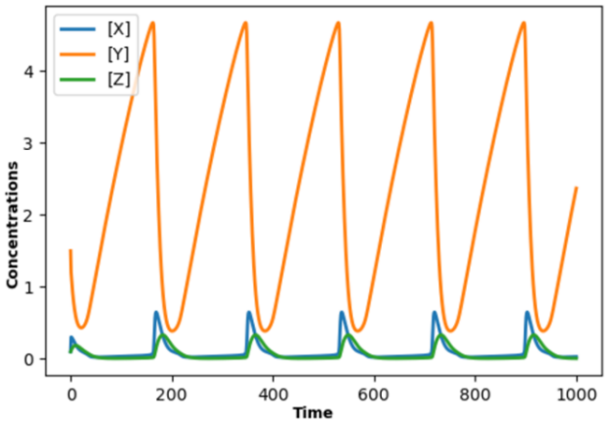


Figura 5.1: Comportamento del modello `BIOMD0000000184`.

la Petri net corrispondente alla CRN, mostrata in Figura 5.2. Qui è, inoltre, possibile osservarne la rappresentazione in linguaggio DOT che è stata generata e salvata su file.

L'insieme dati generati in questo modo vanno a costituire un dataset, che sarà, quindi composto, dalla Petri net e dai dati generati a tempo di simulazione di ogni modello studiato.

```
1   digraph "BIOMD0000000184.xml"{
2       X [shape=circle];
3       Y [shape=circle];
4       Z [shape=circle];
5       R1 [shape=box];
6       R1 -> X [arrowhead=vee, label=1.0];
7       R2 [shape=box];
8       X -> R2 [arrowhead=vee, label=1.0];
9       R3 [shape=box];
10      Y -> R3 [arrowhead=vee, label=1.0];
11      R3 -> X [arrowhead=vee, label=1.0];
12      Z -> R3 [arrowhead=dot];
13      R4 [shape=box];
14      X -> R4 [arrowhead=vee, label=1.0];
15      R4 -> Y [arrowhead=vee, label=1.0];
16      R5 [shape=box];
17      Y -> R5 [arrowhead=vee, label=1.0];
18      R5 -> X [arrowhead=vee, label=1.0];
19      R6 [shape=box];
20      R6 -> Z [arrowhead=vee, label=1.0];
21      X -> R6 [arrowhead=dot];
22      R7 [shape=box];
23      Z -> R7 [arrowhead=vee, label=1.0];
24  }
```

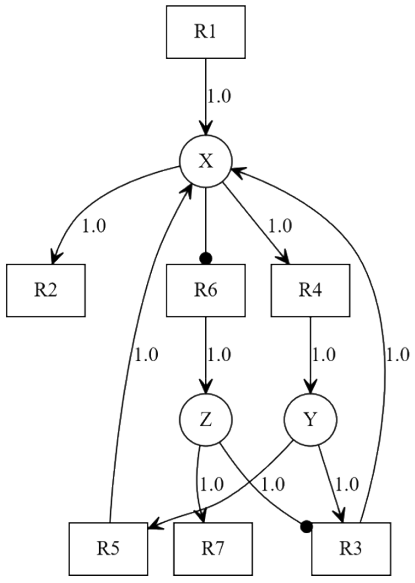


Figura 5.2: Petri net rappresentante una rete di reazioni chimiche.

Capitolo 6

Conclusioni

È stato creato un package pronto per l'installazione e l'uso da parte di terzi che presenta le caratteristiche descritte nei precedenti capitoli. Questo permette lo studio di modelli biologici e biochimici utilizzando un approccio *in silico* così da automatizzare l'analisi e la simulazione. Quando possibile, vengono eseguite analisi sulla struttura delle CRN utilizzando teoremi presenti in letteratura in quanto ciò presenta dei vantaggi rispetto allo svolgimento dello stesso studio ricorrendo, però, a simulazioni. In versioni future potranno essere introdotti alcuni miglioramenti. Uno di questi consiste nello spostare l'esecuzione delle simulazioni sulle GPU. Ciò sarà possibile grazie al compilatore apposito che stanno sviluppando per la libreria libRoadRunner, il quale permetterà di usare GPU Nvidia [7]. Un altro miglioramento riguarda, invece, la selezione dei modelli. In alcuni di essi le regole sono utilizzate in modo simile alle funzioni definite dall'utente e, per questo motivo, potrebbero essere considerati modelli da studiare. In un futuro lavoro questi potrebbero essere analizzati in modo più approfondito così da poterli riconoscere dagli altri, i quali, invece, presentano regole che rendono il modello non adatto al nostro scopo.

La partecipazione a questo tirocinio è stata un'esperienza molto istruttiva e ha arricchito molto il mio bagaglio culturale. Mi ha permesso di conoscere e di avvicinarmi alla disciplina della bioinformatica. Ho avuto l'occasione di apprendere concetti base riguardo la biochimica, come le reazioni chimiche, gli approcci che possono essere usati per studiarle, la chimica cinetica, ma anche concetti di carattere meno generale, come i teoremi formulati da Feinberg presenti in letteratura. Mi è stato permesso di applicare nozioni apprese durante il mio percorso formativo allo studio di reti di reazioni chimiche. Ad esempio, il concetto di grafo è stato usato per formalizzare un sistema così da poterlo rappresentare come matrice di incidenza o come liste di adiacenza. Inoltre, ho conosciuto nuovi formalismi e linguaggi per la loro rappresentazione, come le Petri nets descritte mediante il linguaggio DOT. Ho avuto l'opportunità di approcciarmi a nuovi strumenti software come la repository BioModels, il linguaggio Python e, più in particolare, le librerie libRoadRunner e libSBML.

Tra le difficoltà maggiori incontrate compaiono l'apprendimento dei concetti

di biochimica e l'uso delle librerie Python utilizzate, specialmente libRoadRunner. Ciò è legato alla gestione delle eccezioni e dei valori generati dalle funzioni messe a disposizione da parte di questa libreria. Tutte le funzioni usate sono state analizzate e testate sui modelli indicati nel Capitolo 5 così da poter gestire tutti i possibili casi che si sono riscontrati.

Bibliografia

[1] Andy Somogyi, Kyle Medley. *libRoadRunner*. URL: http://libroadrunner.org.

[2] *Apache License 2.0*. URL: https://choosealicense.com/licenses/apache-2.0/.

[3] Python Packaging Authority. *Packaging Python Projects*. URL: https://packaging.python.org/tutorials/packaging-projects/.

[4] Daniela Besozzi. *Systems Biology. Approccio bottom-up: modelli deterministici e stocastici*. URL: https://homes.di.unimi.it/besozzi/sysbio/lezioni/DetStocX2.pdf.

[5] Bove, P.; Milazzo, P.; Micheli, A. and Podda, M. ≪Prediction of Dynamical Properties of Biochemical Pathways with Graph Neural Networks≫. In: *Proceedings of the 13th International Joint Conference on Biomedical Engineering Systems and Technologies: BIOINFORMATICS,* 2020, pp. 32–43. DOI: 10.5220/0008964700320043.

[6] *Come scegliere una licenza per i propri progetti*. URL: https://www.gnu.org/licenses/license-recommendations.html.

[7] Endre T. Somogyi, Jean-Marie Bouteiller, James A. Glazier, Matthias König, Kyle Medley, Maciej H. Swat, Herbert M. Sauro. *libRoadRunner: A High Performance SBML Simulation and Analysis Library*. URL: https://arxiv.org/pdf/1503.01095.pdf.

[8] Martin Feinberg. ≪Chemical reaction network structure and the stabilit of complex isothermal reactors–I. The Deficiency Zero and Deficiency One Theorems≫. In: *Chemical Engineering Science* 42.10 (1987), pp. 2229–2268. DOI: https://doi.org/10.1016/0009-2509(87)80099-4.

[9] Benjamin J Bornstein; Sarah M Keating; Akiya Jouraku; Michael Hucka. ≪LibSBML: An API Library for SBML≫. In: *Bioinformatics* 24.6 (2008), pp. 880–881. DOI: 10.1093/bioinformatics/btn051.

[10] Maxim Lavrentovich e Sheryl Hemkin. ≪A mathematical model of spontaneous calcium(II) oscillations in astrocytes.≫ In: *Journal of theoretical biology* 251.4 (2008), pp. 553–60. DOI: 10.1016/j.jtbi.2007.12.011.

[11] Rahuman S Malik-Sheriff et al. «BioModels — 15 years of sharing computational models in life science». In: *Nucleic Acids Research* 48.D1 (gen. 2020). gkz1055, pp. D407–D415. ISSN: 0305-1048. DOI: 10.1093/nar/gkz1055. eprint: https://academic.oup.com/nar/article-pdf/48/D1/D407/31698010/gkz1055.pdff. URL: https://doi.org/10.1093/nar/gkz1055.

[12] Matt McCormick. *Making a Python Package*. URL: https://python-packaging-tutorial.readthedocs.io/en/latest/setup_py.html.

[13] Michael Hucka (Chair), Frank T. Bergmann, Claudine Chaouiya, Andreas Dräger, Stefan Hoops, Sarah M. Keating, Matthias König, Nicolas Le Novère, Chris J. Myers, Brett G. Olivier, Sven Sahle, James C. Schaff, Rahuman Sheriff, Lucian P. Smith, Dagmar Waltemath, Darren J. Wilkinson, Fengkai Zhang. *The Systems Biology Markup Language (SBML):Language Specification for Level 3 Version 2 Core*. URL: http://sbml.org/Special/specifications/sbml-level-3/version-2/core/release-2/sbml-level-3-version-2-release-2-core.pdf.

[14] Paolo Milazzo. *Computational Modelling of Biological Systems*.

[15] Lucia Nasti. «Verification of Robustness Property in Chemical Reaction Networks». In: Università di Pisa, 2020. Cap. 3.

[16] Python. *Python 3.8.3 documentation: Input and Output*. URL: https://docs.python.org/3/tutorial/inputoutput.html.

[17] Sarah Keating, Frank Bergmann, Ben Bornstein, Akiya Jouraku, Lucian Smith. *libSBML Python API*. URL: http://sbml.org/Software/libSBML/5.18.0/docs/python-api/index.html.

[18] Graphviz - Graph Visualization Software. *Node, Edge and Graph Attributes*. URL: http://www.graphviz.org/doc/info/attrs.html#k:arrowType.

[19] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduzione agli algoritmi e strutture dati*. College. McGraw-Hill Education, 2010. ISBN: 9788838665158.

[20] Treccani. *in silico*. URL: http://www.treccani.it/enciclopedia/in-silico_%28Lessico-del-XXI-Secolo%29/.

[21] Wikipedia. *DOT (graph description language)*. URL: https://en.wikipedia.org/wiki/DOT_(graph_description_language).

[22] Wikipedia. *Gillespie algorithm*. URL: https://en.wikipedia.org/wiki/Gillespie_algorithm.

[23] Wikipedia. *In silico*. URL: https://it.wikipedia.org/wiki/In_silico.

[24] Wikipedia. *Rete di Petri*. URL: https://it.wikipedia.org/wiki/Rete_di_Petri.

[25] Wolfgang Reisig, W. Brauer, G. Rozenberg, A. Salomaa. *Petri Nets: An introduction*. Springer-Verlag, 2012. ISBN: 9783642699689.