Rapport de stage de L3

Analyse de la dynamique des modèles biologiques par programmation logique

Léo-Paul Delsaux[†]

Encadrant du stage
Maxime Folschette[‡]

Juin-Août 2022

[†]ENS de Lyon
[‡]Équipe Bio-Computing, labortaoire CRIStAL, CNRS de Lille

Table des matières

1 Introduction

La biologie s'intéresse à l'étude des systèmes vivants ou composant le vivant (gènes, cellules, écosystèmes...). La bioinformatique est l'utilisation de méthodes informatiques pour aider la biologie. Un des domaines de la bioinformatique s'apelle la *biologie des systèmes* et consiste à proposer des modèles et des méthodes informatiques et mathématiques pour représenter et étudier de tels systèmes.

Ainsi, des études biologiques permettent de déterminer comment certains gènes interagissent entre eux, ce qui peut être représenté mathématiquement sous la forme d'un graphe. De plus, ces interactions peuvent être vues comme des transitions entre les sommets de notre graphe, ce qui se rapproche alors plus d'un automate. La représentation du modèle étudié lors de ce stage est explicité dans le chapitre Modèle.

L'étude de la dynamique de systèmes biologiques lève plusieurs problèmes tels que l'identification d'attracteurs, les bifurcations ou encore la connexité entre deux états globaux. Dans ce rapport je vais discuter de la recherche d'attracteurs dans des réseaux d'automates asynchrones : il s'agit d'un ensemble d'états duquel on ne peut pas s'échapper et minimal au sens de l'inclusion.

Ce qui sera étudié dans ce rapport ne le sera que partiellement : on n'entrera pas en détail dans le code présenté, mais on se penchera simplement sur les grandes lignes et sur les points techniques qu'il arbore.

2 Answer Set Programming

L'Answer Set Programming (ASP) est un paradigme de programmation logique comparable à Prolog. Ces dernières décennies, ASP s'est trouvé être puissant pour traiter des modèles biologiques, permettant de parcourir un grand nombre de configurations rapidement. ASP peut énumérer facilement les ensembles solutions à un problème qu'on lui encode : c'est un paradigme très efficace pour la combinatoire. Nous allons ici présenter brièvement son fonctionnement, et plus précisément, les outils qui ont été utiles pour mon stage.

2.1 Termes

On commence par définir un terme (ou atome) en ASP :

1. terme simple :
    - un entier relatif (0, 1, 42, -35, ...)
    - une constante qui démarre par un enderscore ou une lettre minuscule, et est suivie par des lettres/chiffres/enderscore (v0us, av35, p3rdu, au, j3u, __sisi__, ...)
    - une chaîne de caractères sous la forme "ma_chaine_de_caracteres" ("gf43ha43gG", "42", ...)

- une variable identique grammaticalement à la constante, mais dont la première lettre doit être majuscule (Je, Suis, Une, Variable, ...)
- un enderscore symbolisant une variable sans nom
- $\#sup$ et $\#inf$ qui sont des constantes définies par ASP

2. une fonction est de la forme : constante(terme, terme, ..., terme) avec un nombre de termes fini qui peut être nul, et qu'on appelle 'arité' (f(23), je(suis(une(fonction(42, true)), et(pas_moi))), ...)

3. un tuple de la forme : (terme, terme, ..., terme) avec un nombre de termes fini qui ne peut être nul, et qu'on appelle 'arité' ((37, oui), (), (1,h(-4),0,1), ...)

2.2 Règles et modèles d'un programme

Un *programme d'ensemble solutions (= answer set program)* est un nombre fini de règles de la forme :

$$a_0 \text{ :- } a_1, ..., a_m, not\ a_{m+1}, ..., not\ a_n.$$

avec $n \geq m \geq m \geq 0$. a_0 est un atome ou $\perp$ (Bottom/le Faux) et représente ici la **tête** de la règle. Les $a_1, ..., a_n$ sont des atomes et représente quant à eux le **corps** de la règle, et le symbole "*not*" représente la négation par l'échec. Cette règle se lit intuitivement : si les atomes $a_1, .., a_m$ sont tous vrais et qu'aucun des atomes $a_{m+1}, ..., a_n$ n'est vrai, alors a_0 est vrai.

Si $n = m = 0$, cela signifie que a_0 est vrai. Dans ce cas, on parle d'un fait, et on ne doit pas renseigner ":-". D'une autre part, si $a_0 = \perp$, on parle de contrainte : comme $\perp$ ne peut jamais être vrai, si le corps de la règle est vrai, cela invalide la solution actuelle. On ne renseigne pas $\perp$ pour la tête, on laisse une tête vide au niveau du code.

Cette notion de "solution actuelle" se formalise de la façon suivante : on parle d'une **interprétation** I en tant qu'ensemble fini d'atomes propositionnels. Une règle r définie comme ci-dessus est *vraie dans* I si et seulement si :

$$\{a_1, ..., a_m\} \subseteq I \wedge \{a_{m+1}, ..., a_n\} \cap I = \emptyset \Rightarrow a_0 \in I$$

Si toutes les règles d'un programme P sont vraies dans une même interprétation I, alors on dit que I est un **modèle** de P. Lorsque l'on déclare un problème P en ASP, le solveur nous renverra en sortie tout les modèles possibles pour P.

2.3 Variables

Si une variable apparaît dans un atome de la tête, elle doit également être dans la queue. Chaque instance d'une variable va être groundée par clingo afin de trouver tous les modèles à notre problème. Prenons un petit exemple, voici un problème encodé avec 2 règles :

parentOf(jenny, charles).
parentOf(mary, jenny).

Ce programme contient 2 faits : Charles est un parent de Jenny, et Jenny est un parent de Mary. C'est ainsi que l'on souhaite comprendre les termes "parentOf(jenny, charles)" et "parentOf(mary, jenny)". Si on souhaite désormais spécifier que Charles est un grand parent de Mary, on peut le faire en ajoutant le terme : "grandparentOf(Mary, Charles)". Cependant, ASP est capable de faire bien mieux pour cela. En effet, on peut généraliser la notion de grand parent comme étant le parent d'un parent de la façon suivante :

grandparentOf(X, Z) :- parentOf(X, Y), parentOf(Y, Z).

Pour chaque valeurs possible de X, Y, et Z, le grounding d'ASP va nous créer des règles associées. Ainsi, les 27 règles dont grandparentOf(mary, mary) :- parentOf(mary, mary), parentOf(mary, mary), ou encore grandparentOf(charles, mary) :- parentOf(charles, jenny), parentOf(jenny, mary) vont être créées par cette façon de procéder. Ceci n'est pas du tout un problème : le solveur d'ASP va simplement regarder si les atomes des queues sont vraies ou pas. Si ce n'est pas le cas, comme le faux implique tout, les règles seront vraies. Si c'est le cas, la tête sera alors mise à vraie. Sachant qu'une seule de ces 27 possibilités mène à la véracité d'un nouvel atome, on obtiendra en sortie d'ASP les faits suivants :

SATISFAIBLE
parentOf(jenny, charles)
parentOf(mary, jenny)
grandparentOf(mary, charles)

2.4 Agrégats

Les agrégats forment le dernier outil d'ASP que j'ai utilisé dans le cadre de mon stage. Il s'agit d'un moyen de sélectionner un certain nombre d'atomes parmi un ensemble, et de les mettre à vrai. Ils se structurent suivant l'exemple suivant :

$$0\ \{f(A,B) : g(A)\}\ 2.$$

La **borne inférieure**, située à gauche des accolades, et la **borne supérieure** définissent l'intervalle discret du nombre d'atomes que l'on va mettre à vrai. Si l'une de ces deux bornes est omise, ASP comprend qu'il s'agit de la borne maximale (ou minimale) possible.

3 Formalismes d'automates

On définit ce qu'est un réseau d'automates asynchrones proprement.

3.1 AAN

Très simple à utiliser en pratique (pour la bio).

3.2 Traduction en ASP

Un bout de code à expliquer en ASP.

4 Dynamique

état local, sémantique, état global, état stable, domaines de pièges, attracteurs + les lemmes et tout ça.

5 Révision du code

6 Conclusion

7 Bibliographie