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

Ce rapport fait écho au chapitre de Ben Abdallah et al. [2022]. Mon stage a démarré par la lecture complète de ce chapitre, nécessaire pour traiter le sujet. Je vais donc commencer mon rapport en introduisant les notions majeures de ce chapitre.

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 expliqué dans la partie "Formalismes d'automates et leur dynamique".

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.

Answer Set Programming (= ASP) est un paradigme de programmation logique particulièrement efficace pour résoudre des problèmes combinatoires. Mon stage s'est concentré sur l'étude d'ASP adin de chercher les attracteurs parmi le graphe des états globaux de notre réseau. J'ai également utilisé un peu se scripting Python, utile pour filtrer les ensembles solutions.

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

ASP fonctionne à l'aide de déclarations de faits, de règles, de prédicats contenant un ou plusieurs arguments tels que : *parentOf(jenny, charles)*, ce qui nécessite tout d'abord que j'introduise la notion de **terme** (ou **atome**). On définit un terme de la manière suivante :

1. terme simple :
   — un entier relatif (0, -35, 42)

— une constante qui démarre par une lettre minuscule(v0us, lis35, quatre, ex3mpl35)
— une chaîne de caractères encadrée de guillemets ("ma_chaine_de_caracteres", "42")
— une variable identique grammaticalement aux constantes, mais dont la première lettre est majuscule (N0u5, S0mm35, D35, V4r14bl35)
— un (ou plusieurs) underscore symbolisant une variable sans nom (_, __)

2. une fonction (qu'on appellera également prédicat) est de la forme : constante(t_1, t_2, ..., t_k) avec un nombre de termes (qui sont les arguments) fini qui peut être nul (f(23), je_suis_une_fonction(42, true), et_moi_aussi(oui)).
   Ces fonctions n'ont pas vocation à calculer, mais uniquement à déclarer des objets avec les arguments souhaités comme étant vrais.
3. un tuple de la forme : (t_1, t_2, ..., t_k) avec un nombre de termes fini qui ne peut être nul ((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 $n \geq m \geq 0$. a_0 est un atome ou $\bot$ (Bottom/le Faux) et représente ici la **tête** de la règle. Les $a_1, ..., a_n$ sont des atomes et représentent quant à eux le **corps** de la règle, et le symbole "*not*" représente la négation par l'échec. Cela signifie que si l'on ne peut pas montrer que x est vrai, alors *not* x est faux. 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 $a_0 = \bot$, on parle de contrainte : comme $\bot$ ne peut jamais être vrai, si le corps de la règle est vrai, cela invalide la solution actuelle. On ne renseigne pas $\bot$ 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 et que I est maximale, 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 le corps. Chaque instance d'une variable va être groundée par ASP afin de trouver tous les modèles à notre problème. Prenons un 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 grand père de Mary, on peut le faire en ajoutant le terme : "grandparentOf(mary, charles)". Cependant, ASP est capable de faire de la déduction. 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 valeur 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 corps sont vraies ou pas. Si ce n'est pas le cas pour une règle donnée (i.e. l'ensemble des éléments du corps de cette règle est faux), comme $(\bot \Rightarrow x) \Leftrightarrow Vrai$, les règles seront vraies. Si c'est le cas pour une règle donnée, la tête de cette dernière sera alors mise à $Vrai$ afin que l'interprétation vérifie bien le plus de règles possible. Sachant qu'une seule de ces 27 possibilités mène à la véracité d'un nouvel atome, on obtiendra la sortie suivante en ASP :

SATISFIABLE
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 :

$0\ \{coloration(X, Y, Teinte) : couleur(Teinte)\}\ 1$:- abscisse(X), ordonnee(Y).

La **borne inférieure** située à gauche des accolades (ici 0), et la **borne supérieure** (ici 1) définissent l'intervalle discret du nombre d'atomes que le solveur peut 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 (à savoir 0 pour la borne inf et cardinal de l'ensemble défini par les accolades pour la borne sup). Dans cet exemple, on choisira donc une ou 0 couleur parmi celles possibles (telles qu'il existe un atome couleur(Teinte) qui soit Vrai) pour chaque abscisse et ordonnée définies.
Les agrégats permettent de faire des disjonctions de cas.

2.5 Premier exemple : le sudoku

Durant les premières semaines de mon stage, j'ai encodé différents jeux de logique en ASP afin de me familiariser avec le langage. Le jeu le plus simple et connu que j'ai encodé a été le sudoku. Dans cette section, je vais brièvement détailler le fonctionnement de ce code.
On codera une grille de sudoku en ASP en utilisant s, une fonction d'arité 3 prenant X, Y et V comme arguments, où X est la colonne, Y la ligne et V la valeur de la case (X,Y) dans la grille.

% 1..9 est un raccourci pour dire itérer pour les entiers allant de 1 à 9. Ici, on déclare qu'il y

a 9 valeurs dans une fonction d'arité 1 : 'val'

1. $val(1..9).$

   % On déclare également 3 autres faits pour mémoriser où sont situés les bordures de notre grille car le point virgule permet de déclarer plusieurs atomes en un

2. $border(1;4;7).$

   % On ne prend qu'une seule valeur par carré

3. $1\,\{s(X,Y,V) : val(V)\}\,1 \text{ :- } val(X); val(Y).$

   % Une valeur ne peut pas apparaître plusieurs fois dans la même colonne

4. $1\,\{s(X,Y,V) : val(Y)\}\,1 \text{ :- } val(X); val(V).$

   % Ni dans la même ligne

5. $1\,\{s(X,Y,V) : val(X)\}\,1 \text{ :- } val(Y); val(V).$

   % Une valeur ne peut pas apparaître plusieurs fois dans une sous-grille.

6. $1\,\{s(X,Y,V) : val(X), val(Y), X1 <= X, X <= (X1+2), Y1 <= Y, Y <= (Y1+2)\}\,1 \text{ :-}$
   $\hookrightarrow\ val(V); border(X1); border(Y1).$

Une fois que l'on a fait cela, il ne nous reste plus qu'à instancier notre problème sur une grille. Pour cela, on renseigne *s(X,Y,V)* pour chaque case déjà pré-numérotée de la grille à notre code, et le solveur se chargera de nous renvoyer l'intégralité de la grille complétée. Si une grille possède plusieurs solutions, le solveur retournera toutes les solutions. De plus, si la grille ne possède aucune solution, le solveur renverra *UNSATISFIABLE*.

2.6 Scripting Python

En ASP, on peut écrire des morceaux de script en Lua ou en Python. Pour cela, il suffit d'ajouter la commande '#script (mon_langage)', puis de taper son code dans le langage que l'on a choisi, et de finir le script par la commande '#end.'. Le scripting admet trois atouts majeurs :

- L'appel aux fonctions en ASP via la commande '@ma_fonction(et, ses, arguments)', permettant d'effectuer des calculs sous Python et de mettre des variables à vrai en fonction de ce qui a été trouvé dans ces calculs.
- Un contrôle de la résolution permettant le filtrage des ensembles solutions, l'ajout de faits/règles, la demande de grounding et de solving. C'est ce dont je me suis le plus servi durant mon stage.
- Une méthode incrémentale efficace lorsque l'on a besoin d'avoir une notion de temporalité. Ceci permet d'ajouter une notion d'ordre en ASP (les lignes de code peuvent être écrites dans n'importe quel ordre).

2.7 Second exemple : le sokoban

Le sokoban a été le dernier jeu de logique que j'ai encodé en ASP, et le plus complexe à cause de la duplication très couteuse du nombre de coups nécessaires pour la résolution d'une grille.

Le jeu est composé d'une grille possédant des cases qui sont du sol ou du mur. Sur une case représentant le sol, il y a le joueur. Sur plusieurs autres cases de sol il y a des caisses. Sur autant de cases de sol qu'il y a de caisses, on trouve des cases d'arrivée (qui sont également du sol). Le but du jeu est de pousser toutes les caisses sur les cases d'arrivée sachant que le joueur ne peut se déplacer qu'en haut, à gauche, en bas ou à droite, et ne peut pas traverser les murs ni pousser une caisse s'il y a un mur ou une autre caisse derrière celle-ci.

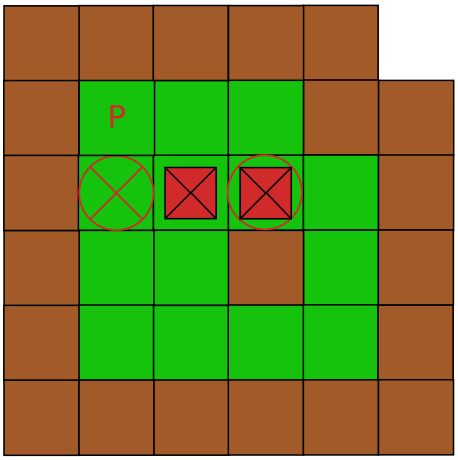

FIGURE 1 – Exemple d'une grille type de sokoban. P symbolise le joueur, les ronds rouges sont les cases d'arrivée, et les carrés représentent les caisses.

En ASP, une grille de sokoban est encodée en spécifiant quelles cases sont des murs, et quelles cases sont des sols avec les prédicats *mur(X, Y)* et *sol(X, Y)*. On renseigne de plus un prédicat *init(X, Y)*, ce qui nous donne la position initiale du joueur, et *caisse_init(Numero_caisse,X, Y)* nous renseignant sur les positions initiales des caisses (le premier argument varie de 1 à n lorsque l'on a n de ces caisses). Enfin, le prédicat *arrivee(X, Y)* symbolisera une case d'arrivée en *(X, Y)*.

J'ai fait différentes versions pour ce jeu. La plus aboutie utilise une méthode incrémentale. Pour l'importer, il nous suffit d'ajouter la commande *#include <incmode>.*. On doit alors définir les sous-programmes **base**, **step(k)** et **check(k)**.

— **base :**
On déclare ici des prédicats *perso(0,X, Y)* et *caisse_a_instant(0,X, Y)* afin de mémoriser la position du joueur à l'étape initiale (0), ainsi que celles des caisses.
Pour notre exemple figure 1, on aura *perso(0,2,5)*, *caisse_a_instant(0,3,4)*, et *caisse_a_instant(0,4,4)*.

— **step(k) :**
On a ici besoin de prédicats *acces_a_instant(k, R, C)* pour mémoriser toutes les cases accessibles depuis la position *(X, Y)* de notre joueur à une étape k donnée. Pour déterminer cela, on va mettre des règles pour chaque direction spécifiant que si une case est accessible à une étape k et qu'une des 4 cases voisines à celle-ci est vide (un sol sans caisse) alors cette case voisine est également accessible. On doit également rajouter une règle pour dire que la case sur laquelle se trouve notre personnage à l'étape choisie est accessible (cas initial pour l'induction).
Pour notre exemple figure 1, toutes les cases de sol (= vertes) ne possédant pas de caisse sont accessibles. On va donc générer 12 prédicats de la sorte pour l'étape 0.
On choisit ensuite un coup pour chaque instant à l'aide d'un agrégat. Si à un instant k on a deux cases de sol c_1 et c_2 sur une même ligne ou colonne et séparées d'exactement une case avec une caisse telles que l'on ait accès à c_1 et qu'il n'y ait pas de caisse en c_2, alors le personnage peut se déplacer jusqu'en c_1, puis pousser la caisse située entre c_1 et c_2 en c_2. On obtient ainsi un coup jouable pour l'instant k. L'agrégat nous sélectionne

alors un seul de ces coups, qui sera sous la forme d'un prédicat *coup_a_instant(k, (R, C), (R2, C2)*.
Dans notre cas figure 1, seuls deux coups sont possibles : *coup_a_instant(1,(3,3),(3,4))* et *coup_a_instant(1,(3,5),(3,4)).*
A partir du choix du coup, on est capable de déduire quel sera la prochaine case du personnage (celle de la caisse que l'on vient de déplacer), et on peut créer les prédicats *perso(k,X,Y)* et *caisse_a_instant(k,X,Y)* associés à l'instant k.
Par exemple si l'on a choisi le premier des deux coups jouables, on en déduira les prédicats *perso(1,3,4), caisse_a_instant(1,3,5)*, et *caisse_a_instant(1,4,4).*
Enfin, on est capable de savoir si on a résolu la grille ou pas à l'aide d'un prédicat *caisse_pas_place(k)* qui est vrai s'il reste une ou plusieurs caisses à placer à l'étape k.
Ici, la caisse en *(3,5)* n'est pas bien placée à l'étape 1, donc on crée le prédicat *caisse_pas_place(1).*

— **check(k) :**
check(k) est le sous-programme qui se charge d'arrêter la recherche : si à une étape k ce sous-programme est satisfiable, alors le programme s'arrêtera et renverra l'ensemble solution courant. Pour vérifier si l'on a trouvé une solution à notre grille, il suffit simplement d'ajouter une contrainte *:- caisse_pas_place(k)*. En effet, s'il existe une caisse non placée le *check*(k) ne sera pas satisfiable.
Dans notre petit exemple, le coup de caisse qu'on a choisi fait peut-être avancer la résolution, mais ne l'a pas achevé.

On récupère alors la liste des coups de caisses à faire, et on peut résoudre notre grille en comblant les coups qui déplaçaient les caisses par ceux nécessaires pour atteindre la prochaine caisse à déplacer.
Pour l'exemple figure 1, ASP nous renvoie la liste de coups suivante : s(1,(3,5),bas), s(2,(5,4),gauche), s(3,(4,4),gauche), s(4,(2,5),bas), s(5,(3,2),haut), s(6,(2,4),droite), s(7,(2,2),haut). *s(k, case, direction)* signifie que le personnage peut aller à l'instant k à la case indiquée, et pousser une caisse dans la direction indiquée.

3 Formalismes d'automates et leur dynamique

Différents modèles existent pour représenter efficacement un système biologique et manipuler sa dynamique : les deux principaux sont les réseaux booléens synchrones de Stuart Kauffman, et les réseaux asynchrones de René Thomas. Dans le cadre de mon stage je ne parlerai que des réseaux asynchrones, et plus particulièrement des réseaux d'automates asynchrone (AAN) Folschette et al. [2015] et Paulevé [2016], qui forment une extension d'une précédente structure appelée "Process Hitting" Paulevé et al. [2014].

3.1 AAN et leur traduction en ASP

Un automate A, dans le contexte de mon stage, sera défini comme étant un ensemble d'états $q_0, q_1, ..., q_{|A|-1}$ avec des transitions dont les étiquettes seront un ou plusieurs états (qu'on appellera également *niveaux*) d'automates externes. Il n'y a donc ni état final, ni état initial. On ne lui donnera pas non plus de mot à lire en entrée puisqu'on va s'intéresser à la dynamique de nos ensembles d'automates. On parlera alors de transition locale : $q_i \xrightarrow{l} q_j$ symbolisera le fait que

l'on peut passer de l'état q_i à l'état q_j si toutes les conditions de l sont vérifiées.
Un réseau d'automates asynchrone est un triplet (Σ, S, T) avec :

— $\Sigma = \{a, b, ...\}$ est un ensemble fini d'automates non vides.

— Si C_a est le nombre d'états d'un automate a, alors $S_a = \{a_0, a_1, ..., a_{C_a-1}\}$ est l'ensemble des **états locaux** de l'automate a. $S = \prod_{a\in\Sigma} S_a$ est l'ensemble fini des **états globaux**, et $LS = \bigcup_{a\in\Sigma} S_a$ représente l'ensemble de tous les états locaux.

— Pour chaque $a \in \Sigma$, $T_a \subseteq \left\{a_i \xrightarrow{l} a_j \in S_a \times \rho(LS/S_a) \times S_a | a_i \neq a_j\right\}$ est l'ensemble des **transitions locales** d'un automate a, avec ρ qui désigne la puissance ensembliste. $T = \bigcup_{a\in\Sigma} T_a$ est l'ensemble des transitions locales du modèle.

Exemple : On représente l'AAN suivant de cette manière :

— $\Sigma = \{a, b, c\}$

— $S_a = \{a_0, a_1, a_2\}$, $S_b = \{b_0, b_1\}$ et $S_c = \{c_0, c_1, c_2\}$

— $T_a = \left\{a_0 \xrightarrow{b_0} a_1, a_0 \xrightarrow{b_1,c_1} a_1, a_1 \xrightarrow{b_1} a_0, a_1 \xrightarrow{b_0} a_2, a_2 \xrightarrow{b_1} a_1\right\}$

$T_b = \left\{b_0 \xrightarrow{c_0} b_1, b_1 \xrightarrow{a_2} b_0\right\}$

$T_c = \left\{c_0 \xrightarrow{b_1} c_1, c_0 \xrightarrow{a_2} c_2, c_1 \xrightarrow{b_0} c_0, c_1 \xrightarrow{a_1} c_2, c_2 \xrightarrow{b_1} c_0\right\}$

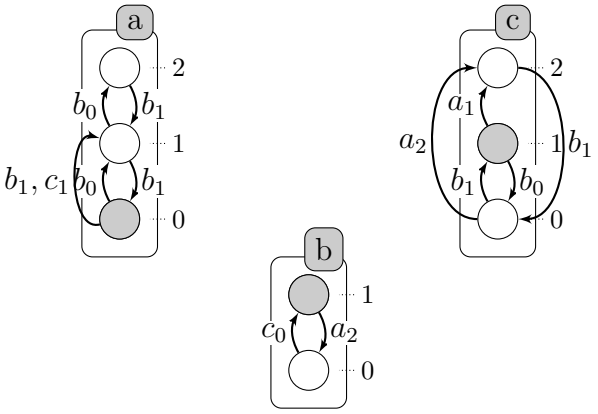

FIGURE 2 – Exemple de réseau d'automates asynchrone

En ASP, on définira un AAN en deux temps :

— On commencera par déclarer chacun de nos automates avec les niveaux (= états) qu'il contient :
automaton_level("a", 0..2).
automaton_level("b", 0..1).
automaton_level("c", 0..2).

— Enfin, les transitions seront encodées via des labels (t1, t2, ...), et on donnera chacune des conditions, ainsi que l'état d'arrivée, via un fait :
condition(t1, "a", 0). target(t1, "a", 1). condition(t1, "b", 0).
condition(t2, "a", 1). target(t2, "a", 2). condition(t2, "b", 0).
condition(t3, "a", 2). target(t3, "a", 1). condition(t3, "b", 1).
condition(t4, "a", 1). target(t4, "a", 0). condition(t4, "b", 1).

```
[...]
condition(t12, "a", 0). target(t12, "a", 1). condition(t12, "b", 1). condition(t12, "c", 1).
```

Nous avons ainsi défini l'automate de notre **exemple** en ASP.

3.2 Sémantiques

Soit $R = (\Sigma, S, T)$ un AAN (défini ci-dessus). On définit la notion de transition jouable :

— On dit qu'une transition locale est **jouable** si toutes les conditions de celle-ci sont vérifiées. On notera P_ζ l'ensemble des transitions locales jouables depuis un état global ζ.

L'état global (a_0, b_1, c_1) *de l'**exemple** admet une seule transition jouable :* $a_0 \xrightarrow{b_1,c_1} a_1$.

La dynamique d'un AAN se définit à l'aide de sa **sémantique**. La sémantique forme l'ensemble des propriétés définissant les **transitions globales jouables** : dans un AAN on s'intéresse à l'évolution globale du réseau.

3.2.1 Asynchrone

Soit $R = (\Sigma, S, T)$ un AAN, et $\zeta \in S$ un état global. L'ensemble des transitions globales jouables depuis ζ pour la sémantique *asynchrone* est donnée par :

$$U^{asyn}(\zeta) = \left\{ \left\{ a_i \xrightarrow{l} a_j \right\} | a_i \xrightarrow{l} a_j \in P_\zeta \right\}$$

De manière informelle : chaque transition locale jouable est une transition globale.

En ASP, on définit préalablement les états globaux à l'aide d'un prédicat de la forme *global_state(Gs)* avec Gs qui s'écrit en ASP à l'aide d'une constante unique (g1_1_0 symbolisera par exemple que "a" est au niveau 1, "b" au niveau 1 et "c" au niveau 0, cf Partie 4.4). On peut ensuite déclarer que deux états globaux sont différents sur un automate à l'aide de la règle suivante :

```
1. different_on(Gs1, Gs2, Automaton) :- % Cet atome est vrai
2.     global_state(Gs1), global_state(Gs2), Gs1 != Gs2, % si deux états globaux diffèrent
3.     automaton(Automaton), % qu'il existe un automate
4.     active_in_g(level(Automaton, LevelI), Gs1), % sur lequel Gs1 vaut I
5.     active_in_g(level(Automaton, LevelJ), Gs2), % et sur lequel Gs2 vaut J
6.     LevelI != LevelJ. % avec I ≠ J
```

Ensuite, on fait comprendre à ASP qu'une transition locale n'est pas jouable si l'une (au moins) de ses conditions n'est pas vérifiée de la façon suivante :

```
1. unplayable(Transition, Gs) :- % Cet atome est vrai
2.     local_transition(Transition), % s'il existe une transition locale
3.     global_state(Gs), % un état global
4.     automaton(Automaton), % et un automate
5.     active_in_g(level(Automaton, LevelI), Gs), % sur lequel Gs vaut I
6.     condition(Transition, Automaton, LevelJ), % alors qu'il devrait valoir J ≠ I
7.     LevelI != LevelJ. % pour que la transition soit jouable
```

On déclare ensuite que deux états globaux différent sur au moins un autre automate qu'*Automaton* avec le terme $not\_equal\_except(Automaton, Gs1, Gs2)$ comme ceci :

```
1. not_equal_except(Automaton, Gs1, Gs2) :- % Cet atome est vrai
```

2. *automaton(Automaton), automaton(Automaton2), % s'il existe deux automates*
3. *Automaton != Automaton2, % qui différent*
4. *global_state(Gs1), global_state(Gs2), % deux états globaux*
5. *different_on(Gs1, Gs2, Automaton2). % tels que* $Gs1 \neq Gs2$ *sur Automaton2*

On peut enfin définir ce qu'est une transition globale jouable à l'aide de nos trois précédentes règles :

1. *playable(Gs1, Gs2) :- % Cet atome est vrai*
2. *global_state(Gs1), global_state(Gs2), % s'il existe deux états globaux* $Gs1$ *et* $Gs2$
3. *automaton(Automaton), local_transition(Transition), % un automate et une transition*
4. *target(Transition, Automaton, LevelJ), % qui fait changer le niveau de l'automate*
5. *not unplayable(Transition, Gs1), % et avec la transition (locale) qui est jouable*
6. *not not_equal_except(Automaton, Gs1, Gs2), % et* $Gs1 = Gs2$ *excepté sur Automaton*
7. *active_in_g(level(Automaton, LevelI), Gs1), % tel que le niveau d'Automaton dans* $Gs1$
8. *active_in_g(level(Automaton, LevelJ), Gs2), % et celui dans* $Gs2$
9. *LevelI != LevelJ. % sont différents*

3.2.2 Synchrone

Soit $R = (\Sigma, S, T)$ un AAN, et $\zeta \in S$ un état global. L'ensemble des transitions globales jouables depuis ζ pour la sémantique *synchrone* est donnée par :

$$U^{syn}(\zeta) = \{u \subseteq T | u \neq \emptyset \wedge \forall a \in \Sigma, (P_\zeta \bigcap T_a = \emptyset \Rightarrow u \bigcap T_a = \emptyset) \wedge (P_\zeta \bigcap T_a \neq \emptyset \Rightarrow |u \bigcap T_a| = 1\}$$

De manière informelle : tous les automates possédant au moins une transition locale jouable doivent changer de niveau.

Pour coder cela en ASP, on peut se servir des atomes *different_on* et *unplayable*.
On ajoute de plus une autre règle : *has_playable(Automaton, Gs, LevelI, LevelJ)* spécifiant que l'on peut faire changer *Automaton* dans Gs du niveau I vers le niveau J.
Enfin, il ne nous reste plus qu'a déclarer (par la négation) quelles sont les transitions globales qui ne sont pas jouables. Pour cela, on essaie pour toutes les paires d'états globaux si on peut faire la transition de l'un à l'autre en créant un terme *not_playable(Gs1,Gs2)* si la transition ne peut pas avoir lieu. On énumère alors les différents cas de figure :

1. si *Automaton* n'admet aucune transition locale jouable dans Gs_1 et que $Gs_1 \neq Gs_2$ sur *Automaton*
2. si *Automaton* admet une (ou plusieurs) transition locale jouable dans Gs_1 et qu'il n'existe pas de transition pour faire passer le niveau d'*Automaton* sur Gs_1 à celui de Gs_2
3. si aucune transition locale n'est jouable depuis Gs_1

alors la transition $Gs_1 \rightarrow Gs_2$ n'est pas jouable.
Enfin, il nous suffit de tester pour toutes les paires d'états globaux distincts si on a *not_playable(Gs1,Gs2)* ou pas. Dans le cas contraire, alors on crée l'atome *playable(Gs1,Gs2)*.

3.2.3 Généralisée

Soit $R = (\Sigma, S, T)$ un AAN, et $\zeta \in S$ un état global. L'ensemble des transitions globales jouables depuis ζ pour la sémantique *généralisée* est donnée par :

$$U^{gen}(\zeta) = \{u \subseteq T | u \neq \emptyset \wedge \forall a \in \Sigma, (P_\zeta \bigcap T_a = \emptyset \Rightarrow u \bigcap T_a = \emptyset) \wedge (P_\zeta \bigcap T_a \neq \emptyset \Rightarrow |u \bigcap T_a| \leq 1\}$$

De manière informelle : tous les automates possédant au moins une transition locale jouable peuvent changer de niveau, et au moins l'un d'entre eux doit changer de niveau.

Pour coder cela en ASP, on prend exactement le code de la sémantique synchrone, en remplaçant le second cas de figure par ceci :

2. si Gs_1 différe de Gs_2 sur *Automaton* et qu'il n'existe pas de transition pour faire passer le niveau d'*Automaton* sur Gs_1 à celui de Gs_2

Cela permet bien de ne pas prendre une transition par automate qui est jouable puisqu'on va tester des paires d'états globaux qui ont peut-être 3 automates jouables, et si l'on ne fait qu'une seule transition locale cela ne sera pas filtré par notre atome *not_playable(Gs1,Gs2)*.

3.3 Dynamique

Soit U une sémantique. On introduit les dernières notions nécessaires pour la définition d'un attracteur :

- On appelle **état stable** un état global ne possédant aucune transition globale jouable pour U. *Dans l'**exemple**, aucun état global n'est stable pour la sémantique asynchrone (et donc cela est également vrai pour la sémantique synchrone et généralisée).*
- Pour $Gs_1, Gs_2 \in S$, on notera $Gs_1 \rightarrow_U Gs_2$ pour symboliser le fait qu'il existe $u \in U(Gs_1)$ tel que si l'on joue toutes les transitions locales de u, alors Gs_1 devient Gs_2.
- Un **chemin** est une famille finie $(A_k)_{1 \leq k \leq n}$ et ordonnée d'états globaux tels que pour tout $1 \leq k \leq n-1$, $A_k \rightarrow_U A_{k+1}$.
- Un **cycle** est un chemin $(A_k)_{1 \leq k \leq n}$ tel que $A_1 = A_n$.
- Un **domaine de piège** est un ensemble non vide d'états globaux $S_E \in S$ tel que toute transition globale jouable depuis S_E arrive dans un état global de S_E. Plus formellement : $\forall \zeta_1 \in S_E \wedge \forall \zeta_2 \in S, \zeta_1 \rightarrow_U \zeta_2 \Rightarrow \zeta_2 \in S_E$

Enfin, on peut définir ce qu'est un attracteur.
Un ensemble $A \subseteq S$ d'états globaux, avec $|A| \geq 2$ est appellé **attracteur** si et seulement si c'est un domaine de piège minimal en terme d'inclusion ensembliste.
Ben Abdallah et al. [2022] a démontré le résultat mathématique suivant dont on va se servir pour trouver les attracteurs au sein d'un AAN :

Lemme : Les attracteurs d'un AAN sont exactement les domaines de piège cycliques.

4 Cœur de la contribution personnelle

J'ai tout d'abord étudié la version existante du code afin de la comprendre et d'essayer de corriger l'erreur qu'elle contenait. Cette version était fonctionnelle et efficace pour la sémantique asynchrone, mais ne fonctionnait pour la sémantique synchrone que lorsque les attracteurs étaient exactement des cycles simples (chaque état global du graphe produit doit avoir un degré sortant égal à 1), et ne fonctionnait pas pour les autres AAN comme celui-ci :

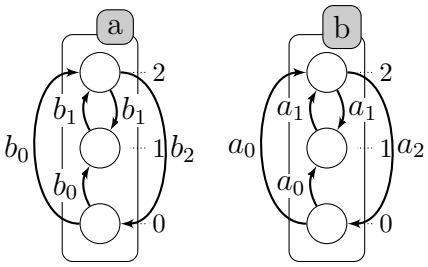

FIGURE 3 – Exemple d'AAN sur lequel le code pré-existant ne trouvait pas l'attracteur

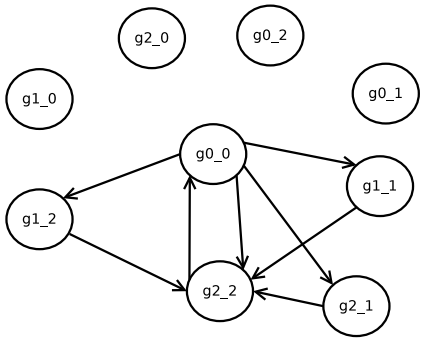

FIGURE 4 – Graphe produit de l'AAN ci-dessus

En effet, l'ensemble $\{(0,0),(1,1),(1,2),(2,1),(2,2)\}$ est un attracteur de cet AAN pour la sémantique synchrone. De plus, il existe deux (même quatre ici) transitions globales jouables depuis l'état global $(0,0)$: on peut aller en $(1,1)$ ou encore en $(2,2)$.

Après avoir identifié le problème au sein du code, j'ai trouvé deux alternatives pour corriger cela : la première consiste à reprendre tout le code, enlever le problème ciblé et corriger cela avec un script Python, et la seconde consiste à considérer les états globaux et à définir les transitions globales jouables dans des fichiers dédiés aux sémantiques. Je vais donc détailler deux manières de trouver les attracteurs d'un AAN en ASP.

4.1 Code pré-existant

On explique tout d'abord comment générer des chemins d'une longueur définie dans un AAN.

```
% On mémorise quelques informations : noms des automates et des transitions locales
automaton(Automaton) :- automaton_level(Automaton, _).
local_transition(Transition, Automaton) :- target(Transition, Automaton, _).
local_transition(Transition) :- target(Transition, _, _).

% On définit ensuite les étapes de notre chemin
step(0..n). % Ici n est une constante qu'on définit lors de la compilation (souvent 10)
1 {main_cycle_length(N) : step(N), N >0 } 1.% La longueur du cycle principal
cycle_step(0..N) :- main_cycle_length(N).% On nomme les étapes du cycle principal
after_cycle_step(N+1..n) :- main_cycle_length(N).% différemment de celles d'après

% On choisit ensuite un état initial à l'aide d'un agrégat
1 { active(level(Automaton, Level), 0) : automaton_level(Automaton, Level) } 1 :-
```

```
9.          automaton(Automaton).

% On calcule les transitions locales non jouables pour chaque étape
10. unplayable(Transition, Step) :-
11.         active(level(Automaton, LevelI), Step),
12.         condition(Transition, Automaton, LevelJ),
13.         LevelI != LevelJ, step(Step).

% On trouve les automates qui ont au moins une transition jouable (sémantique synchrone)
14. has_playable(Automaton, Step) :-
15.         not unplayable(Transition, Step),
16.         local_transition(Transition, Automaton),
17.         step(Step).

% On sélectionne une transition à jouer pour chaque automate si possible
18. 1 {played(Transition, Step) :
19.         not unplayable(Transition, Step),
20.         local_transition(Transition, Automaton)
21. } 1 :- has_playable(Automaton, Step).

% Contrainte : on doit jouer au moins une transition locale à chaque étape
22.  :- 0 { played(_, Step) } 0, step(Step).

% Maintenant que l'on a choisi notre coup, on en déduit le changement à faire.
23. change(Transition, Automaton, LevelI, LevelJ, Step) :-
24.         played(Transition, Step),
25.         target(Transition, Automaton, LevelJ),
26.         condition(Transition, Automaton, LevelI).

% On change le niveau actif s'il y a un changement dans Automaton
27. active(level(Automaton, LevelK), Step + 1) :-
28.         change(_, Automaton, _, LevelK, Step),
29.         Step <n.

% On garde le niveau actif s'il n'y a pas de changement dans Automaton
30. active(level(Automaton, LevelK), Step + 1) :-
31.         not change(_, Automaton, _, _, Step),
32.         active(level(Automaton, LevelK), Step),
33.         step(Step), Step <n.
```

Je viens ici d'expliquer comment faire pour générer tous les chemins de longueur n avec la sémantique synchrone. Cependant, pour qu'un de ses chemins nous intéresse, il faut qu'il respecte les 3 contraintes suivantes :

- avoir un cycle de longueur N (lorsque *main_cycle_length(N)* est vrai)
- tout les états globaux du chemin visités après l'étape N (lorsque *main_cycle_length(N)* est vrai) doivent être des éléments du cycle
- toutes les transitions globales jouables depuis chacun des éléments du cycle doivent arri-

ver dans un autre élément de ce cycle (= domaine piège)

Les deux premières conditions se vérifient assez aisèment.

Pour la première, on crée un prédicat *different_states_on(Step1, Step2, Automaton)* nous permettant de déduire si deux états globaux atteints à $Step_1$ et $Step_2$ sont différents ou pas. On en déduit alors un prédicat *same_state(Step1, Step2)*, et il ne nous reste qu'à ajouter la contrainte *:- not same_state(0, N), main_cycle_length(N).*

Quant à la seconde, il nous suffit de créer un prédicat *valid_state_after_main_cycle(Step2)* vrai lorsqu'il existe une étape $Step_1$ dans le cycle principal et une étape $Step_2$ en dehors du cycle principal telles que l'on ait l'atome *same_state(Step1, Step2)* qui soit vrai. Ce prédicat devant être toujours vrai pour chaque $Step_2$, on rajoute finalement la contrainte suivante :
:- not valid_state_after_main_cycle(Step1), after_cycle_step(Step1).

La troisième contrainte contenait la raison du bug du code (cf début partie 4). Comme la modélisation actuelle des choses ne mémorise pas quels sont les coups jouables pour la sémantique choisie (cela permettait d'avoir un code qui fonctionne pour toutes les sémantiques en scindant le fichier ASP en deux : recherche des attracteurs, et sémantique, ce qui permettait d'avoir un seul solveur d'attracteur, universel pour toutes les sémantiques), cela n'était pas possible. Deux options étaient alors envisageables pour résoudre ce problème : faire un script ASP dans laquelle je rajoute la dernière contrainte en introduisant une notion d'état global ; ou bien utiliser Python pour résoudre ce problème lors du solving de clingo.

4.2 Résolution de la troisième contrainte en Python

Je ne vais pas rentrer en détail dans la manière dont on peut réussir à récupérer des atomes lors du grounding et du solving d'ASP en Python : on admettra que cela est faisable. Nous avons donc en Python une *boucle for* qui va, à chaque itération, avoir un ensemble d'atomes tous vrais tel que l'interprétation qu'ils forment satisfasse toutes les règles de notre programme ASP.

Lors de la première itération de cette boucle, il est nécessaire de mémoriser toutes les transitions, les conditions, ainsi que les cibles de ces transitions. On introduit donc quelques ensembles pour stocker cela en mémoire.

1. *condition_transitions = {} # de la forme [transition][automata] = level*
2. *target_transitions_automaton_name = {} # de la forme [transition] = automaton_name*
3. *target_transitions_automaton_level = {} # de la forme [transition] = level*
4. *transitions_already_done = False*

Ensuite, dès le début de la *boucle for* magique, on récupère toutes les informations nécessaires pour remplir ces ensembles, et on modifie la variable booléenne *transitions_already_done* à True afin de ne plus faire cela lors des autres itérations de la boucle.
On en profite également pour créer un ensemble *cur_set* de la forme *[step_number][automata_name] = active_level* qui viendra mémoriser les états globaux de chaque étape de l'ensemble solution courant.

Ensuite, on crée un ensemble *playable* qu'on va venir remplir avec les transitions locales jouables

depuis chacun des états globaux de l'attracteur courant et chaque automate de ce dernier. Il est de la forme *playable[global_state][automata_name][transition] = level* où 'level' est le niveau de l'automate après la transition. Pour remplir ce dernier, il faut qu'on teste pour chaque transition locale si elle est jouable pour l'état global courant ou pas en regardant pour chaque condition de la transitions si elle est vérifiée ou pas. Si elle est bien jouable, on l'ajoute au dictionnaire.

On se sert de l'ensemble *playable* pour créer une liste de listes de listes d'entiers *list_of_list*.Pour faire simple, chacune de ses sous-listes représente un état global de l'attracteur courant. Chacune des sous-listes de ces états globaux représente un automate, et chacune des sous-listes de cet automate correspond à l'ensemble des niveaux dans lesquels il peut se retrouver après une transition locale jouable depuis l'état global. Si aucune transition locale n'est jouable, la sous-liste de l'automate en question sera vide. Pour remplir cette liste, on procède de la façon suivante :

- Pour chaque état global de l'attracteur courant on crée une liste *list_for_current_state* (des automates)
- Pour chaque automate de cet état global on crée une liste *list_for_automaton*(des niveaux atteignables)
- Si aucune transition locale n'est jouable pour l'automate, on le laisse à son niveau
- Sinon pour chaque transition locale jouable on détermine le niveau après transition de l'automate et on l'ajoute à la liste des niveaux atteignables de l'automate
- on ajoute la liste des niveaux atteignables de l'automate à l'état global, puis la liste de l'état global à *list_of_list*

L'utilisation d'une liste de listes est très pratique puisqu'il nous suffit alors d'effectuer un produit cartésien de listes pour en déduire les listes de tout les états globaux atteignables depuis notre attracteur courant. C'est exactement ce qu'on calcule dans *list_of_product* : la liste des produits cartésiens de chaque coup possible pour un état global.

Enfin, il nous suffit de vérifier que chaque élément de list_of_product est bien un élément du cycle principal. De cette façon, notre attracteur vérifiera bien la troisième condition, et sera valide. Si un au moins des éléments de *list_of_product* n'est pas dans l'attracteur, alors il ne s'agit en réalité pas d'un attracteur : on peut s'en échapper.

C'est ici que l'on termine la *boucle for* magique.

Les résultats de cette version sont plus qu'appréciables. En effet, j'ai utilisé la même batterie de tests que Maxime Folschette pour la version de laquelle je suis parti, et les durées de résolution se sont avérées toutes aussi bonnes que celles de la sémantique asynchrone.

4.3 Légère modification pour sémantique généralisée

En reprenant le code présenté ci-dessus, j'ai également fait une version pour la sémantique généralisée (la version de laquelle je suis parti fonctionnant très bien pour la sémantique asynchrone, je n'ai rien à dire dessus). En effet, la différence avec la sémantique synchrone est que l'on peut faire changer nos automates ou pas, mais qu'au moins un de tous les automates doit changer de niveau lorsque l'on recherche quelles sont les transitions globales jouables. Concrètement, il faut modifier la sémantique dans le code pré-existant en ASP :

```
[...]
% On sélectionne 0 ou 1 transition à jouer pour chaque automate
18. 0 {played(Transition, Step) :
19.         not unplayable(Transition, Step),
20.         local_transition(Transition, Automaton)
21. } 1 :- has_playable(Automaton, Step).
[...]
```

On doit ensuite modifier dans le code python le fait que *list_for_automaton* soit initialement vide : pour chaque automate, on peut soit rester dans cet automate, soit faire une des transitions trouvés. On va donc initialiser cette liste de cette façon :
list_for_automaton = [global_state[sorted_automata_names.index(automaton_name)]] (où *sorted_automata_names* est une liste des noms d'automates triée par ordre lexicographique). Ensuite, on rajoute à *list_for_automaton* tous les autres niveaux atteignables pour l'état global courant (tout comme la sémantique synchrone).

4.4 Étude des états globaux

L'autre solution que je voyais pour résoudre le problème qu'ont rencontré mon maître de stage et ses collègues était de déclarer des prédicats *global_state(Gs)* représentant les états globaux du réseau. Cela permet de déclarer dans des fichiers pour les sémantiques quelles sont les transitions locales jouables en utilisant un prédicat *playable(Gs1, Gs2)* signifiant que l'on peut aller depuis Gs_1 en Gs_2.

Le principal défaut de cette manière de procéder est que cela crée de nombreux atomes et demande à ASP de les garder en mémoire pendant toute sa résolution. De plus, déterminer si pour toute paire d'états globaux il existe une transition se fait en $O(k^2)$, avec $k = \prod_{a \in \Sigma} C_a$, alors que si l'on calcule les transitions jouables à chaque étape, on ne surcharge pas la mémoire du solveur. C'est à priori ce qui rend mon programme obsolète en terme de complexité.

Les explications d'ASP que j'ai faites pour la partie 3.2 sur les différentes sémantiques sont en réalité mes propres fichiers pour les trois sémantiques.

Dans cette section, je compte expliquer comment fonctionne mon fichier ASP principal, ainsi que comment je m'y suis pris pour déclarer des états globaux sans modifier la manière d'écrire un automate (en ASP on ne peut pas créer de tuples ayant une arité variable en fonction d'un prédicat).

J'ai d'abord défini deux fonctions en Python permettant de créer une constante pour un état global représenté par une liste-ASP (un objet que j'introduis par la suite) : ce sera la fonction *work(g)*, et de déterminer les niveaux des automates actifs de ces constantes : ce sera la fonction *determine_active_level(a,g)* . Je ne détaillerais pas le code de ces fonctions.

— *work(g)* prend un état global encodé sous forme de liste-ASP et renvoie une constante unique en ASP représentant l'état global en entrée.

— *determine_active_level(a,g)* prend un automate a sous la forme d'une chaîne de caractères,

et un état global g sous la forme de sa constante en ASP, et renvoie un prédicat "level(a, niveau_de_a)" en ASP.

La structure du code ASP reprend fortement celle de la version pré-existante, c'est pourquoi je ne vais pas la mettre intégralement.

Après avoir introduit quelques informations (noms d'autoamtes, transitions locales, étapes du chemin) comme c'est fait dans les 7 premières lignes de la version pré-existante, on introduit 2 prédicats supplémentaires utiles pour définir un ordre parmi les automates. On utilise alors une fonction de clingo : #count {E} permettant de compter le nombre d'éléments dans l'ensemble E.

```
8. nb_automaton(N) :- N = #count { A : automaton(A) }.
9. nb_level(A, N) :- automaton(A), N = #count { L : automaton_level(A, L) }.
10. enum(1..N) :- nb_automaton(N).
```

À l'aide des prédicats *enum(k)* que l'on vient juste de créer, on peut désormais définir un ordre pour les automates :

```
11. 1 { moment(A, N) : enum(N) } 1 :- automaton(A).
12. 1 { moment(A, N) : automaton(A) } 1 :- enum(N).
13.  :-
14.      moment(A, N), moment(A2, N2),
15.      automaton(A), automaton(A2),
16.      enum(N), enum(N2),
17.      N <N2, A <A2.
```

Une fois cet ordre défini, on est capable de créer une liste-ASP pour encoder chacun des états globaux. Le $(N-k)^{ieme}$ élément de cette liste sera un prédicat nb(automaton, level) avec automaton étant k^{ieme} au niveau de l'ordre défini précédemment (on empile dans la liste d'abord les premiers éléments, c'est pour ça qu'on a trié par ordre anti-lexicographique).

```
18. build_list(0, empty).
19. build_list(N+1, (nb(A, 0..K-1), L)) :-
20.      build_list(N, L),
21.      enum(N+1),
22.      moment(A, N+1),
23.      nb_level(A, K).
```

On utilise ensuite nos fonctions Python pour travailler avec des constantes plus agréables à lire que des grosses listes ASP.

```
24. gs(L) :- build_list(N, L), nb_automaton(N).
25. global_state(@work(gs(L)), L) :- gs(L).
26. global_state(Gs) :- global_state(Gs, _).
27. active_in_g(@determine_active_level(Automaton, L), Gs) :-
28.      automaton(Automaton),
29.      global_state(Gs,L).
```

Avec tout ces éléments, il est alors très facile de poursuivre l'algorithme et de déclarer les 3 contraintes de l'attracteur comme on l'avait fait dans la version pré-existante : on choisit un état global à l'étape initiale (step 0), pour chaque étape on choisit un coup jouable en connaissant l'état dans lequel on se trouve à cette étape. Puis on détermine quel est l'état global de l'étape suivante, et on teste ensuite les trois contraintes avec les états globaux. La troisième contrainte est alors très simple à encoder maintenant que l'on a mémorisé les coups jouables dans la sémantique :

```
30. also_playable(Gs1, Gs2, Step) :-
31.     playable(Gs1, Gs2),
32.     not played(Gs1, Gs2, Step),
33.     active_g(Gs1, Step),
34.     step(Step).

35. evolves_in_main_cycle(Gs1, Gs2, Step1, Step2) :-
36.     playable(Gs1, Gs2),
37.     active_g(Gs1, Step1),
38.     active_g(Gs2, Step2).

39. :- also_playable(Gs1, Gs2, Step), not evolves_in_main_cycle(Gs1, Gs2, Step, _).
```

Les résultats de cette version sont décevants. Avec les mêmes tests que précédemment, je n'ai réussi à obtenir que 2 (sur 7) solutions en moins de 100s, ce qui n'est vraiment pas bon. Maxime Folschette et moi-même pensons que cela est dû à la trop grosse quantité de mémoire d'ASP gardé pour de trop nombreux prédicats.

5 Conclusion et pistes pour la suite

Durant mon stage j'ai élaboré deux versions pour résoudre le problème rencontré avec l'ajout de la troisième contrainte. J'ai présenté dans ce rapport la solution qui fait appel à un script Python, ainsi que la version modifiée pour traiter la sémantique généralisée. J'ai très brièvement parlé de la solution nécessitant la création d'états globaux en ASP.

Mes performances sur la batterie de test qu'a utilisé Maxime Folschette pour la version de laquelle je suis parti se sont avérées toutes aussi bonnes que celles de la sémantique asynchrone. Les résultats de ces tests sont disponibles sur le github de mon stage, juste ici. Les performances de ma seconde version y sont également. On voit notamment à quel point elle n'est pas efficace en temps.

Concernant cette seconde version, je pense que l'on peut grandement améliorer son efficacité en combinant mon code avec de l'incrémental. L'idée serait de ne considérer qu'un certains nombres d'états globaux : au début on les considère tous, puis, lorsque l'on trouve un attracteur, on regroundre le programme en enlevant tout les états globaux de l'attracteur (en les mettant tous à faux).

Une autre idée que j'avais en tête était, lorsque l'on trouvait un cycle qui n'était pas un attracteur, de remplacer tout les états de ce cycle par un seul représentant, dont les transitions globales entrantes et sortantes seront celles de chacun des états globaux du cycle. On travaillerait ensuite avec des représentants et des états globaux comme s'il s'agissait du même objet (si on a un représentant dans un autre cycle, on l'agrandira en rajoutant les autres éléments du cycle). Je pense que combiner cette idée avec de l'incrémental pourrait également être intéressant : à chaque cycle trouvé on effectue de nouveau un grounding en mettant à jour nos représentants.

Je ne suis pas sûr de ces pistes, mais c'est ce sur quoi j'essaierais personnellement de me pencher pour poursuivre le travail.

Enfin, je souhaite remercier l'ENS de Lyon qui m'a proposé ce stage, Maxime Folschette pour son encadrement, les personnes au sein de l'équipe BioComputing, mes quelques collègues stagiaires de bureau ainsi que mes quelques relecteurs anonymes.

Mon code est disponible ici : https ://github.com/Leopoulpinator/Attractors-in-AAN.

Références

Emna Ben Abdallah, Maxime Folschette, and Magnin Morgan. Analyzing Long-Term Dynamics of Biological Networks with Answer Set Programming. working paper or preprint, July 2022. URL `https://hal.archives-ouvertes.fr/hal-03735849`.

Maxime Folschette, Loïc Paulevé, Morgan Magnin, and Olivier Roux. Sufficient conditions for reachability in automata networks with priorities. *Theoretical Computer Science*, 608 :66–83, 2015.

Loïc Paulevé. Pint, a static analyzer for dynamics of automata networks. In *14th International Conference on Computational Methods in Systems Biology (CMSB 2016)*, 2016.

Loïc Paulevé, Courtney Chancellor, Maxime Folschette, Morgan Magnin, and Olivier Roux. Analyzing large network dynamics with process hitting. *Logical Modeling of Biological Systems*, pages 125 – 166, 2014.