Université de Mons
Faculté des Sciences
Département d'Informatique
Service d'Informatique Théorique

Résolution de jeux de sûreté joués sur graphes

Directrice : M^me^ Véronique BRUYÈRE

Rapporteurs : M^r^ Clément TAMINES
M^r^ Gaëtan STAQUET

Mémoire réalisé par
Florent HUYLENBROECK

en vue de l'obtention du grade de
Master en Sciences Informatiques

Année académique 2021-2022

Table des matières

Chapitre 1

Introduction

En informatique théorique, le problème du model-checking consiste à vérifier qu'un système informatique, exprimé sous la forme d'un modèle mathématique, satisfait une spécification. Par exemple, vérifier que le système n'atteint jamais un état de deadlock, qu'une variable du système ne se voit jamais attribuer une valeur nulle, ou qu'une requête reçoit toujours une réponse. Les modèles utilisés peuvent varier : les comportements du système informatique peuvent être modélisés par un automate acceptant des mots infinis, tandis que la spécification peut être modélisée par une formule de logique temporelle LTL.
Cependant, plutôt que de vérifier qu'un tel système satisfait une spécification, on peut aller plus loin en envisageant la synthèse de contrôleur. Ainsi, dans le but de modéliser les interactions d'un système informatique avec son environnement, on peut considérer un graphe infini dont les sommets sont partagés entre le système et l'environnement. Les interactions entre les deux partis sont alors représentées par un chemin infini dans ce graphe, tel qu'en tout sommet du système (resp. de l'environnement), c'est lui qui décide quel arc suivre à partir de ce sommet.
Le système peut être perçu comme jouant un jeu en s'opposant à l'environnement. Son but est de satisfaire la spécification, peu importe ce que fait ce dernier. Pour gagner ce jeu, le système va devoir se munir d'une stratégie gagnante. Cette stratégie est ce que l'on appelle un programme de contrôle.
La synthèse de contrôleur revient donc à construire, quand c'est possible, une stratégie gagnante pour le système contre l'environnement pour un objectif donné.

Dans ce mémoire, nous allons étudier ce problème de synthèse pour un type de jeu spécifique : les jeux de sûreté. L'objectif de ce jeu est d'éviter au système de passer par certains sommets du graphe. Autrement dit, d'empêcher le système informatique d'atteindre certaines configurations.

Quand le graphe sur lequel le jeu est joué est de taille finie, il existe des algo-

rithmes simples qui indiquent si le système peut y parvenir et qui calculent les stratégies gagnantes pour ce dernier. Quand le graphe est infini, par contre, il n'existe pas d'algorithme exact. Nous allons étudier une méthode de calcul de stratégie gagnante, proposée par [2], consistant à transformer le jeu de base en un ensemble d'automates, pour ensuite en déduire une série de problèmes de satisfiabilité. Ces derniers, s'ils sont satisfaisable, pourront donner une stratégie gagnante pour le système.

Après avoir introduit les notions nécessaires, nous allons commencer par présenter un algorithme existant pour les jeux joués sur graphes finis. Ensuite, nous allons étudier la méthode introduite ci-dessus pour les jeux inifinis. Une implémentation sera finalement proposée pour cette méthode et la complexité de tous ces algorithmes sera discutée.

Chapitre 2

Jeux joués sur graphes

Avant de proposer des méthodes de résolution pour les jeux de sûreté, nous allons d'abord introduire les notions liées aux jeux à duration infinie, joués par deux joueurs sur un graphe, les stratégies et nous intéresser à la condition de victoire pour deux types de jeu en particulier : les jeux de sûreté et les jeux d'atteignabilité.

2.1 Arènes

Une *arène* est un graphe $A = (V_0, V_1, E)$ composé de deux ensembles disjoints, non-vides de sommets V_0 et V_1, avec $V_0 \cup V_1 = V$, et d'un ensemble d'arcs $E \subseteq V \times V$. De plus, chaque sommet d'une arène doit posséder au moins un successeur.

La figure 2.1 représente une arène. Les sommets carrés appartiennent à V_0 et les sommets ronds à V_1. L'ensemble E correspond aux arcs reliant ces sommets.

2.2 Coups, parties et objectifs

Au début de la partie, un *pion* est placé sur un sommet du graphe. Ce pion est un marquage d'un sommet de l'arène qui va être modifié tour à tour par les joueurs. Dans la suite de ce rapport, *déplacer le pion* référera à l'action de retirer le marquage du sommet courant, et de marquer un autre sommet du graphe.
Le jeu est joué par deux *joueurs* numérotés j_0 et j_1 qui déplacent le pion le long des arcs de l'arène. Le joueur qui déplace le pion est le joueur à qui appartient le sommet courant où se trouve le pion. Un *coup* est l'action d'un joueur de déplacer le pion le long d'un arc du graphe, depuis un sommet lui appartenant, vers un de

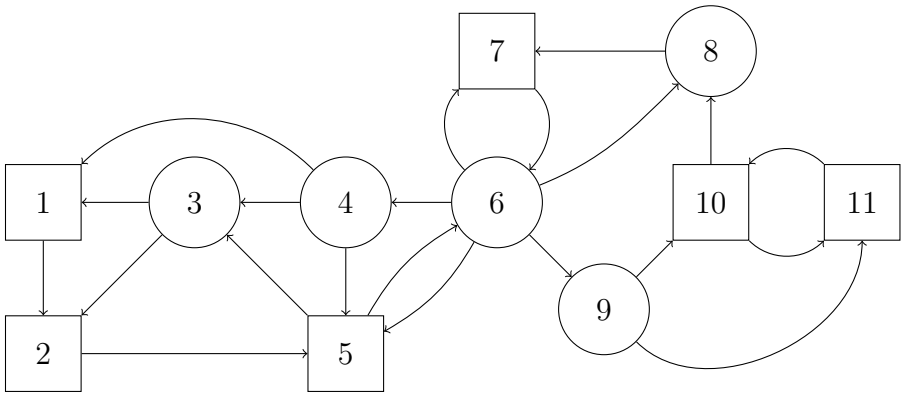


FIGURE 2.1 – Exemple d'une arène

ses successeurs.
Une séquence infinie de coups forme une *partie*.

Définition 2.2.1. *Une* partie *d'un jeu est une séquence infinie* $\pi = v_0 v_1 ...$ *où* $\forall i \in \mathbb{N}, v_i \in V$, *et* $(v_i, v_{i+1}) \in E$.

L'*objectif* Ω d'un jeu joué sur graphes est ce qui va définir la condition de victoire des joueurs. Dans ce rapport, nous allons nous intéresser à deux types de jeux : les jeux de sûreté et les jeux d'atteignabilité. L'objectif pour ces deux types de jeux est défini par un sous-ensemble $F \subseteq V$. Dans ces jeux, les deux joueurs jouent selon des objectifs opposés.

2.2.1 Jeux de sûreté

Un jeu de sûreté est défini par un tuple $\mathfrak{G} = (A, I, F)$ avec A une arène, $I \subseteq F$ un ensemble de sommets de départ et $F \subseteq V$ un sous-ensemble de sommets dits *sûrs*. L'objectif pour le joueur j_0 d'un jeu de sûreté est que tous les sommets visités lors de la partie soient des sommets sûrs. L'objectif pour le joueur j_1 est donc qu'au moins un sommet hors de F soit visité.

Définition 2.2.2. *Soit* $\mathfrak{G} = (A, I, F)$ *un jeu de sûreté, une partie* $\pi = v_0 v_1 ...$ *est gagnante pour le joueur* j_0 *si* $\forall i \in \mathbb{N}, v_i \in F$

2.2.2 Jeux d'atteignabilité

Un jeu d'atteignabilité est défini par un tuple $\mathfrak{G} = (A, I, F)$ avec A une arène, $I \subseteq V \setminus F$ un ensemble de sommets de départ et $F \subseteq V$ un sous-ensemble de sommets *à atteindre*. L'objectif pour le joueur j_0 d'un jeu de sûreté est qu'au

moins un sommet de F soit visité au cours de la partie. L'objectif du joueur j_1 est que seuls des sommets hors de F soient visités.

Définition 2.2.3. *Soit $\mathfrak{G} = (A, I, F)$ un jeu d'atteignabilité, une partie $\pi = v_0v_1...$ est gagnante pour le joueur j_0 si $\exists i \in \mathbb{N}, v_i \in F$*

Les jeux d'atteignabilité sont en dualité avec les jeux de sûreté. En effet, on remarque que l'objectif du joueur j_0 (resp. j_1) d'un jeu de sûreté est similaire à celui du joueur j_1 (resp. j_0) pour un jeu d'atteignabilité.

2.3 Stratégies et ensembles gagnants

Les coups des joueurs sont décidés par la *stratégie* adoptée par ces derniers. Une stratégie est une fonction $s_{j_n} : V^*V_n \to V$ qui indique vers quel sommet le joueur j_n va déplacer le pion depuis le sommet courant $v \in V_n$ selon la séquence de déplacements précédents.
Une stratégie peut être *sans mémoire* (ou *positionnelle*) $s_{j_n} : V_n \to V$ si elle ne prend en compte que le sommet actuel où se trouve le pion.

Une stratégie pour le joueur j_n est *gagnante à partir d'un sommet v* si une partie jouée selon cette stratégie mène à une victoire du joueur j_n.

Définition 2.3.1. *Une partie $\pi = v_0v_1...$ est dite* jouée selon une stratégie s_{j_n} *(à partir d'un sommet $v_0 \in V$) si $\forall i \in \mathbb{N}, v_i \in V_n, v_{i+1} = s_{j_n}(v_0v_1...v_i)$*

La notion de stratégie gagnante nous permet de définir un *ensemble gagnant*.

Définition 2.3.2. *Soit $\mathfrak{G} = (A, I, F)$ un jeu, avec $A = (V_0, V_1, E)$, l'*ensemble gagnant *pour le joueur j_n, $W_n \subseteq V$, est l'ensemble $W_n = \{v \in V \mid j_n$ possède une stratégie gagnante à partir de $v\}$*

Chapitre 3

Cas fini

Afin de résoudre les jeux de sûreté joués sur des graphes, nous allons distinguer les arènes possédant un nombre fini de sommets de celles en possèdant un nombre infini.

Définition 3.0.1. *Une* arène finie *est une arène $A = (V_0, V_1, E)$ pour laquelle V_0 et V_1 sont des ensembles finis.*

Dans cette section, nous allons définir la notion d'attracteur, et l'appliquer afin de calculer les ensembles gagnants des arènes finies, et les stratégies gagnantes des deux joueurs. Nous allons aussi proposer un algorithme qui calcule ces attracteurs.

3.1 Résolution via les attracteurs

Une méthode pour calculer les ensembles gagnants des jeux de sûreté et d'atteignabilité se base sur le principe d'attracteur.

Définition 3.1.1. *Soit un jeu $\mathfrak{G} = (A, I, F)$ avec $A = (V_0, V_1, E)$ une arène finie et F un sous-ensemble de sommets, tel que $F \subseteq V$, soit $i \in \mathbb{N}$, le i^e attracteur pour le joueur j_n est l'ensemble :*
$Attr^i_{j_n}(F) = \{v \in V \mid$ *le joueur j_n peut forcer une visite d'un sommet de F depuis v en $\leq i$ déplacements*$\}$

Afin de construire cet objet en incrémentant la valeur de i, [3] nous donne la formule de construction par induction suivante :

$$\begin{aligned} Attr^0_{j_n}(F) &= F \\ Attr^{i+1}_{j_n}(F) &= Attr^i_{j_n}(F) \\ &\quad \cup \{v' \in V_n \mid \exists (v, v') \in E : v \in Attr^i_{j_n}(F)\} \\ &\quad \cup \{v' \in V \setminus V_n \mid \forall (v, v') \in E : v \in Attr^i_{j_n}(F)\} \end{aligned} \tag{3.1}$$

L'intuition derrière cette formule est la suivante :
Initialement, le joueur j_n ne peut forcer une visite d'un sommet de F que depuis un sommet de F. On a donc que l'attracteur de départ, $Attr^0_{j_n}(F)$ ne contient que les sommets de F.
Ensuite, en incrémentant le nombre de coups, autrement dit la valeur de i, on va considérer l'ajout des sommets qui sont des prédécesseurs des sommets de l'attracteur courant, car ceux-ci mettront, dans le pire des cas, un coup de plus à atteindre un sommet de F. Donc, si un sommet est prédécesseur d'un sommet de l'attracteur sans en faire partie lui-même, il y a deux possibilités (correspondant aux deux ensembles unis à $Attr^i_{j_n}(F)$ dans la formule 3.1). Soit le sommet appartient à j_n et ce sera à lui de jouer un coup à partir de ce sommet. Il pourra ainsi décider de s'approcher d'un sommet de F. Il ne faut donc qu'un seul successeur dans l'attracteur courant pour être ajouté à l'attracteur suivant. Par contre si le sommet appartient au joueur opposé à j_n, alors il faut s'assurer que peu importe le coup qu'il joue, il se rapproche d'un sommet de F. Il est donc nécessaire que tous les successeurs du sommet soient dans $Attr^i_{j_n}(F)$ pour que le sommet soit ajouté à l'attracteur suivant.

Par cette construction, on obtient une séquence d'attracteurs $Attr^0_{j_n}(F) \subseteq Attr^1_{j_n}(F) \subseteq ...$ laquelle sera fixe à partir d'une certaine itération $k \leq |V|$ vu que V est un ensemble fini et qu'à chaque itération, au moins un sommet de V est ajouté à l'attracteur. On notera $Attr_{j_n}(F) = \bigcup_{i=0}^{|V|} Attr^i_{j_n}(F)$

Théorème 3.1.1. *Pour un jeu d'atteignabilité, cette construction de l'attracteur pour j_0 vers F donnera l'ensemble gagnant W_0 de j_0, ainsi que la stratégie gagnante positionnelle des deux joueurs.*

Preuve. On a que $Attr_{j_0}(F) \subseteq W_0$ car
- $\forall v \in Attr^{i+1}_{j_0} \cap V_0, v$ possède un successeur dans $Attr^i_{j_0}$.
- $\forall v \in Attr^{i+1}_{j_0} \cap V_1$, tous les successeurs de v sont dans $Attr^i_{j_0}$.
- $Attr^0_{j_0}(F) \subseteq F$

Donc j_0 peut gagner la partie à partir de tous les sommets de $Attr_{j_0}(F)$. Sa stratégie gagnante est, pour chaque coup depuis un sommet de $Attr^{i+1}_{j_0}(F)$, de déplacer le pion vers un sommet dans $Attr^i_{j_0}(F)$ afin de se rapprocher progressivement de $Attr^0_{j_0}(F) \subseteq F$, ce qui est possible par la manière dont est construit l'attracteur. Cette construction explique aussi que le joueur opposé sera forcé d'en faire autant. Cette stratégie est donc gagnante pour j_0 depuis chaque sommet de l'attracteur.

Pour montrer que $W_0 \subseteq Attr_{j_0}(F)$, il faut montrer que j_0 ne peut pas gagner la partie à partir d'un sommet hors de $Attr_{j_0}(F)$, autrement dit que j_1 peut forcer le pion à rester en dehors de $Attr_{j_0}(F)$ depuis tout sommet hors de $Attr_{j_0}(F)$.

Soit un sommet $v \in V_1 \setminus Attr_{j_0}(F)$, alors v possède au moins un arc (v, v') avec $v' \notin Attr_{j_0}(F)$, sinon on aurait $v \in Attr_{j_0}(F)$. La stratégie gagnante pour j_1 dans ce cas est de déplacer le pion le long de cet arc afin de rester hors de l'attracteur.
Soit un sommet $v \in V_0 \setminus Attr_{j_0}(F)$, alors tous les arcs v mènent vers un sommet hors de $Attr_{j_0}(F)$, sinon on aurait $v \in Attr_{j_0}(F)$. j_0 ne peut donc pas entrer dans l'attracteur depuis ce sommet et ne peut donc pas forcer de visite d'un sommet de F depuis ce sommet. La stratégie gagnante de j_1 dans ce cas est de suivre n'importe lequel de ces arcs. Ces deux cas étant exhaustifs, et ayant montré que j_0 est contraint de rester hors de l'attracteur dans ces deux cas, on a bien que $W_0 \subseteq Attr_{j_0}(F)$
L'inclusion étant vérifiée dans les deux sens, on en déduit $W_0 = Attr_{j_0}(F)$. □

Ces deux cas exhaustifs nous permettent aussi de déduire la stratégie gagnante du joueur j_1. A chaque coup, il doit déplacer le pion vers un sommet hors de l'attracteur.

Ainsi, nous avons montré que l'on peut construire l'ensemble gagnant W_0 du joueur j_0 pour un jeu d'atteignabilité en utilisant le principe d'attracteur. On obtient aussi immédiatement l'ensemble gagnant $W_1 = V \setminus W_0$ du joueur j_1. Chaque joueur peut gagner la partie à partir de chaque sommet de leur ensemble gagnant respectif en adoptant les stratégies énoncées ci-dessus.
Un jeu de d'atteignabilité joué selon ces stratégies sera toujours gagnant pour le joueur j_0 si $I \subseteq W_0$.
De plus, la dualité entre un jeu d'atteignabilité et un jeu de sûreté nous permet d'énoncer le théorème suivant :

Théorème 3.1.2. *Cette méthode de résolution pour les jeux d'atteignabilité permet aussi de résoudre les jeux de sûreté.*

Preuve. Soit un jeu de sûreté $\mathfrak{G} = (A, F)$ avec A une arène finie. On construit $Attr_{j_1}(V \setminus F)$, autrement dit la liste de sommets depuis lesquels j_1 peut forcer une visite d'un sommet hors de F. En adoptant la même stratégie que j_0 dans un jeu d'atteignabilité, cet attracteur donne l'ensemble gagnant de j_1 pour un jeu de sûreté. De manière analogue, j_0, en adoptant la stratégie du joueur j_1 du jeu d'atteignabilité, ne pourra gagner le jeu de sûreté que depuis les sommets hors de cet attracteur. □

La figure 3.1 représente, en gris, l'ensemble gagnant du joueur j_0 d'un jeu d'atteignabilité joué sur l'arène de la figure 2.1, avec $F = \{1, 2, 11\}$.

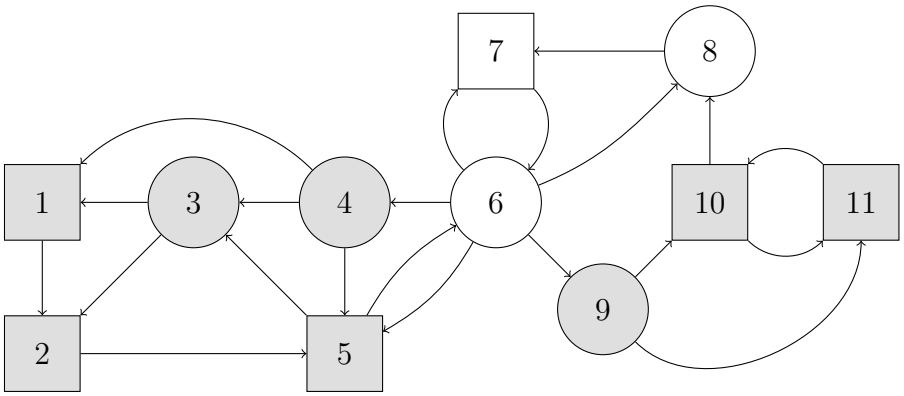


FIGURE 3.1 – Illustration d'un ensemble gagnant

3.2 Implémentation

3.2.1 Algorithme

Nous allons maintenant proposer un algorithme de calcul du i^e attracteur. Pour faciliter la lecture, l'algorithme sera découpé en 3 phases suivies d'explications.

Algorithm 1 Attracteur

Entrées **G** : Graphe, structure de données composée d'un tableau à deux dimensions $predecessors$ de prédécesseurs (la liste de prédécesseurs d'un noeud i est stockée à la i^e entrée du tableau) et une liste $players$ (le i^e noeud appartient au joueur dont le numéro figure en i^e entrée de $players$).
F : Liste de numéros de sommets.
p : Numéro du joueur.
i : Nombre d'itération pour la construction de l'attracteur, une valeur négative calculera l'attracteur complet.

Sortie $Attr_p^i(F)$

1: **procedure** ATTRACTOR(G, F, p, i)
2: $out\_degrees \leftarrow$ tableau de taille $|G|$ ▷ Pré-traitement
3: **for** j allant de 0 à $|G| - 1$ **do**
4: **if** $G.players[j] \neq p$ **then**
5: **for** $pred$ in $G.predecessors[j]$ **do**
6: $out\_degrees[pred] \leftarrow out\_degrees[pred] + 1$
7: **end for**
8: **end if**
9: **end for**

L'algorithme commence par une phase de pré-traitement au cours de laquelle on va calculer le *demi-degré extérieur* (le nombre d'arcs sortants) de chaque noeud n'appartenant pas au joueur p. Pour cela, on initialise une liste $out\_degrees$ de la taille du nombre de noeuds du graphe. Ensuite, on parcourt le tableau des prédécesseurs du graphe $G.predecessors$. On incrémente l'indice de $out\_degrees$ correspondant à chaque noeud rencontré dans ce tableau car s'il est prédécesseur d'un autre noeud, alors un arc en sort pour aller vers celui-ci.

```
10:     attractor ← tableau de taille |G|                    ▷ Initialisation
11:     for index in F do
12:         attractor[index] ← 1
13:     end for
14:     attractor_new ← F
```

On initialise ensuite le tableau $attractor$ qui va indiquer quels sommets sont marqués comme appartenant à l'attracteur courant. On y marque les sommets de F. Cette étape correspond au calcul de $Attr_p^0(F)$.
On initialise ensuite la liste $attractor\_new$. Cette liste va contenir, après chaque itération de la boucle principale, les sommets qui ont été ajoutés à l'attracteur lors de cette itération. On ajoute initialement les sommets de F à cette liste car $Attr_p^0(F)$ est déjà calculé.

```
15:     while attractor_new non vide and i ≠ 0 do        ▷ Calcul de l'attracteur
16:         to_check ← attractor_new
17:         attractor_new ← [ ]
18:         for index in to_check do
19:             for pred in G.predecessors[index] do
20:                 if attractor[pred] = 0 then
21:                     if G.players[pred] = p then
22:                         attractor_new.append(pred)
23:                     else
24:                         out_degrees[pred] ← out_degrees[pred] − 1
25:                         if out_degrees[pred] = 0 then
26:                             attractor_new.append(pred)
27:                         end if
28:                     end if
29:                 end if
30:             end for
31:         end for
32:         for index in attractor_new do
33:             attractor[index] ← 1
34:         end for
35:         i ← i − 1
36:     end while
37:     return attractor
38: end procedure
```

La première étape de la boucle principale est de copier la liste $attractor\_new$

dans une nouvelle liste to_check afin de garder une trace des nouveaux sommets à traiter. On vide ensuite la liste $attractor\_new$ afin de pouvoir éventuellement y ajouter des nouveaux sommets pendant l'itération courante. La boucle principale de cet algorithme se base ensuite sur la construction par induction de $Attr_{j_n}(F)$. On y retrouve les 3 éléments de l'union qui constitue $Attr^{i+1}_{j_n}(F)$ dans 3.1 :

- $Attr^{i}_{j_n}(F)$ se retrouve aux lignes 32-34. A chaque étape, on ne crée pas un nouvel attracteur mais on marque dans $attractor$ les nouveaux sommets présents dans $attractor\_new$.
- $\{v' \in V_n \mid \exists (v, v') \in E : v \in Attr^{i}_{j_n}(F)\}$. Dans la boucle intérieure, lignes 21-22, si un prédécesseur du noeud en cours de traitement appartient au joueur cible, alors il est ajouté à $attractor\_new$ afin d'être ajouté à l'attracteur.
- $\{v' \in V \setminus V_n \mid \forall (v, v') \in E : v \in Attr^{i}_{j_n}(F)\}$. Dans la boucle intérieure, lignes 23-28, si un prédécesseur du noeud en cours de traitement n'appartient pas à p, alors il est ajouté à l'attracteur si tous ses successeurs sont aussi dans l'attracteur. C'est à cette étape que le pré-traitement joue un rôle. A chaque fois qu'un noeud est rencontré dans la liste des prédécesseurs d'un autre noeud, on décrémente la valeur correspondante dans $out\_degrees$. Si cette valeur atteint 0, cela veut dire que tous les successeurs de ce noeud font partie de l'attracteur (car on ne visite les prédécesseurs d'un noeud que s'il a été ajouté à l'attracteur). On peut donc l'ajouter à son tour à l'attracteur.

Le calcul s'arrête quand aucun noeud n'est ajouté à l'attracteur au cours d'une itération ($attractor\_new$ est vide). Cela veut dire que le point fixe de la séquence d'attracteur $Attr^0_p(F) \subseteq Attr^1_p(F) \subseteq ...$ est atteint et que l'attracteur complet a été calculé.

L'algorithme peut aussi retourner le i^e attracteur si on lui passe en entrée une valeur de i positive (et inférieure au nombre d'itération qu'il faut pour atteindre le point fixe). En effet, i intervient dans le calcul de la condition d'arrêt. Celui-ci est décrémenté à chaque nouvel attracteur calculé. Cependant, la condition d'arrêt ne vérifie que si $i \neq 0$. Une valeur négative de i en entrée assurera donc le calcul de l'attracteur complet, car celui-ci ne causera pas l'arrêt de la boucle principale.

3.2.2 Complexité

Considérons un graphe G possédant n noeuds et m arcs. Alors l'algorithme *Attractor* possède une complexité dans le pire des cas en $O(n + m)$.

Preuve. Afin de calculer la complexité totale de l'algorithme, intéressons-nous à la complexité des 3 étapes principales :

— *Pré-traitement* (lignes 1-9)
Le calcul du demi-degré extérieur à l'aide d'une structure de données telle que décrite dans l'en-tête de l'algorithme se fait en temps $O(m)$. En effet, il s'agit d'itérer sur la liste de prédécesseurs et, pour chaque noeud rencontré, incrémenter son demi-degré extérieur. Le graphe possédant m arcs, il y aura au plus m éléments dans la liste des prédécesseurs. Le coût pour chacun de ces prédécesseurs étant en $O(1)$, la complexité de cette opération sera en $O(m)$.

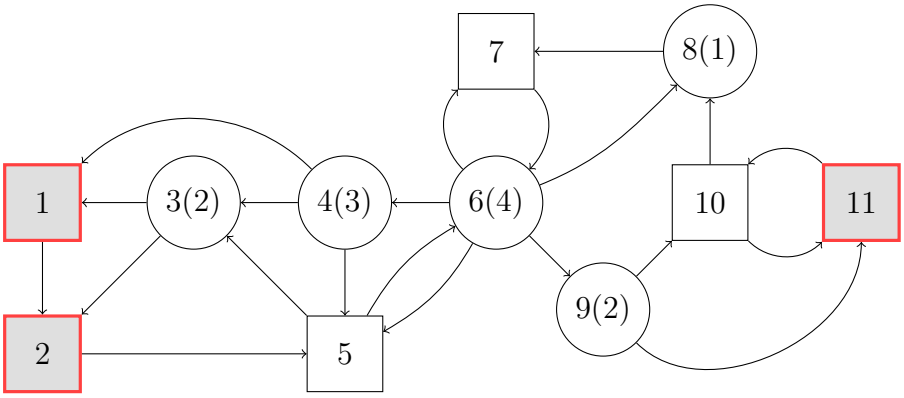
— *Initialisation* (lignes 10-14)
Cette étape se fait en temps $O(n)$ car il y a au plus n noeuds dans le graphe, donc vers lesquels on souhaite construire l'attracteur. Ainsi, au maximum n opérations en $O(1)$ seront effectuées lors de cette étape.

— *Calcul de l'attracteur*
Considérons une valeur de i négative pour le pire des cas.
L'étape du calcul de l'attracteur agit comme un parcours du graphe. En effet, une fois un noeud visité et marqué comme appartenant à l'attracteur, ce noeud ne sera plus visité. Chaque noeud n'est donc visité qu'une seule fois et, dans le pire des cas, tous les noeuds du graphe seront visités.
La condition d'arrêt (ligne 15) est vérifiée au maximum n fois (une fois pour chaque sommet, car il faut au minimum un sommet dans $attractor\_new$ pour satisfaire la condition et, dans le pire des cas, tous les sommets seront ajoutés un par un à l'attracteur). Le coût de cette évaluation est $O(1)$.
Pour chaque sommet traité, on effectue une action en $O(1)$ (ligne 19) afin de récupérer la liste de ses prédécesseurs dans la structure de données G, une autre opération en $O(1)$ (lignes 32-34) afin de marquer ce sommet dans l'attracteur. En comptant aussi la condition d'arrêt et l'opération arithmétique (ligne 35), le coût total par sommet est donc $O(1)$.
Ensuite, pour chaque prédécesseur (autrement dit pour chaque arc), l'algorithme n'effectue que des opérations en $O(1)$: ajouts à une liste (lignes 22 et 26), accès à un tableau via l'indice, comparaisons et opérations arithmétiques (lignes 20, 21, 24 et 25). Le coût total par arc est donc $O(1)$. Nous obtenons un coût total pour le parcours de $n \cdot O(1) + m \cdot O(1) = O(n + m)$.
Il faut ajouter à cela le coût de copier la liste $attractor\_new$ dans to_check. Etant donné que cette liste ne peut contenir qu'une fois chaque sommet à travers toutes les itérations, le coût total des copies vaudra $O(n)$. La complexité de l'étape de calcul est donc $O(n + m + n) = O(n + m)$.

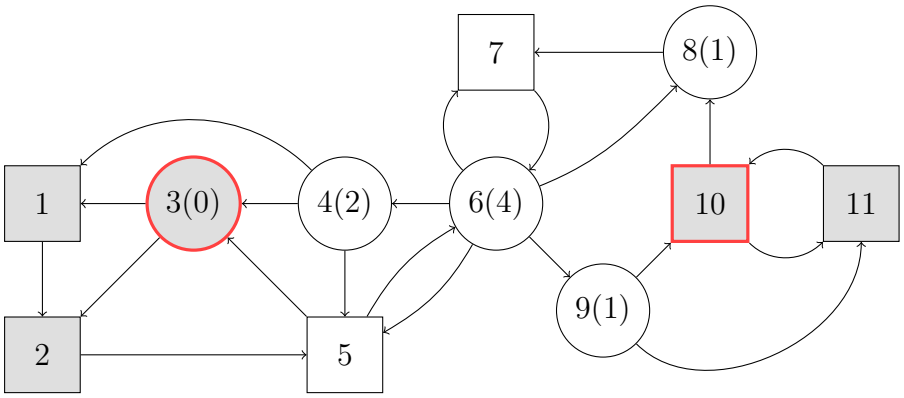
Nous obtenons donc une complexité totale de $O(n+m+(n+m)) = O(2(n+m))$ $= O(n + m)$. □

3.2.3 Exemple

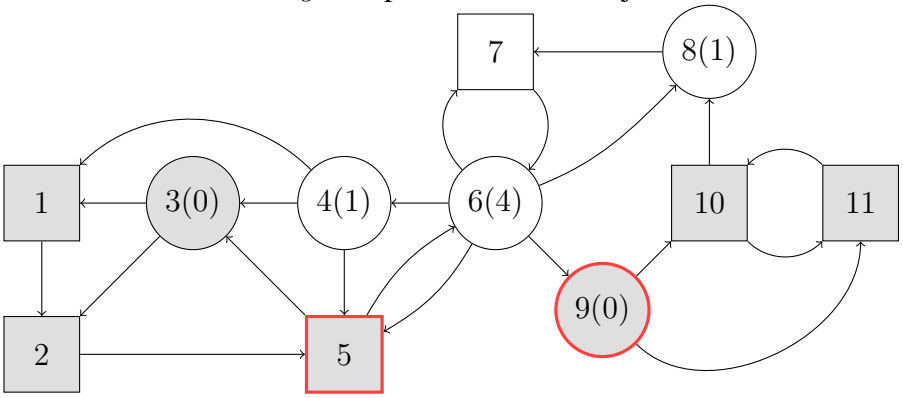
Afin d'illustrer le fonctionnement de l'algorithme, considérons l'arène de la figure 2.1 et calculons l'attracteur pour le joueur j_0 avec $F = \{1, 2, 11\}$, dans le but de calculer les ensembles gagnants des deux joueurs pour un jeu d'atteignabilité. Supposons i négatif afin de calculer l'attracteur complet.
L'étape de pré-traitement sera rendue visuelle en ajoutant aux noeuds de j_1 la valeur correspondant dans le tableau $out\_degrees$. Les noeuds faisant partie de l'attracteur courant $attractor$ seront colorés en gris et ceux étant ajoutés à l'attracteur à l'itération précédente (les noeuds de la liste $attractor\_new$) seront entourés en rouge. Nous obtenons donc, avant l'entrée dans la boucle principale de l'algorithme, la représentation suivante :



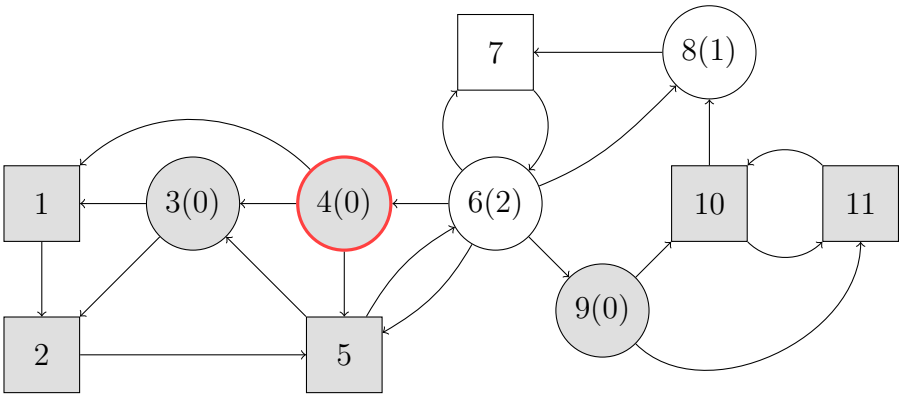
Nous avons donc bien $Attr^0_{j_0}(F) = \{1, 2, 11\}$.
La première itération du calcul de l'attracteur va ajouter les noeuds 3 et 10 à l'attracteur. En effet, la valeur de $out\_degrees$ de 3 va être décrémentée deux fois, une fois en suivant les prédécesseurs de 1 et une autre fois en suivant ceux de 2. Cette valeur atteignant 0, il sera ajouté à l'attracteur. La valeur de $out\_degrees$ de 4 sera aussi décrémentée une fois en partant de 1. Le cas de 10 est plus trivial, il appartient à j_0 et a été rencontré en suivant les prédécesseurs de 11, il est donc ajouté à l'attracteur. La valeur de $out\_degrees$ de 9 est elle aussi décrémentée car le noeud 9 est un prédécesseur de 11. Nous obtenons donc la représentation suivante :



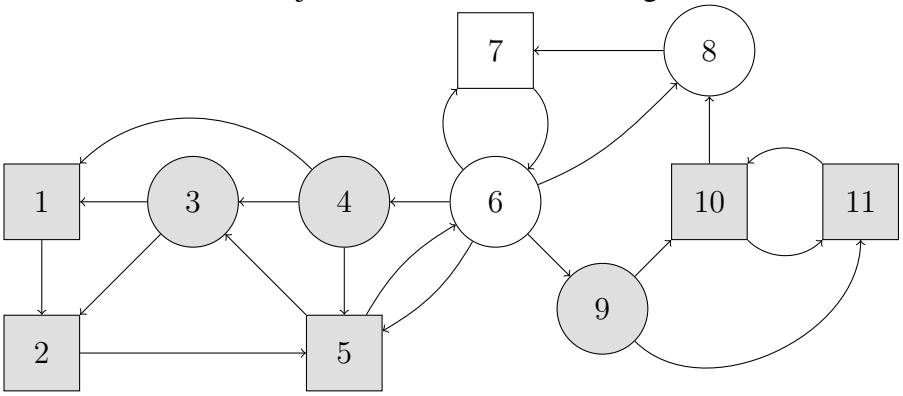
On a $Attr^1_{j_0}(F) = \{1, 2, 3, 10, 11\}$.
5 possède maintenant un successeur dans l'attracteur, il y sera donc ajouté à l'étape suivante. La valeur de $out\_degrees$ de 4 est décrémentée une fois, car il est prédécesseur de 3. Le noeud 9 est atteint une deuxième fois, cette fois depuis 10. Sa valeur de $out\_degrees$ passant à 0, il est ajouté à l'attracteur.



Nous obtenons $Attr^2_{j_0}(F) = \{1, 2, 3, 5, 9, 10, 11\}$
4 est atteint une dernière fois car il est prédécesseur de 5. Sa valeur de out_degree passant à 0, il est ajouté à l'attracteur. Cette même valeur pour 6 est décrémentée deux fois lors de cette itération car il est successeur de 5 et 9.



On obtient $Attr^3_{j_0}(F) = \{1, 2, 3, 4, 5, 9, 10, 11\}$
La valeur de $out\_degrees$ de 6 est décrémentée une fois car il est prédécesseur de 4. Aucun noeud n'est ajouté à $attractor\_new$, l'algorithme s'arrête.



L'attracteur final calculé par l'algorithme est donc $Attr^4_{j_0}(F) = Attr^3_{j_0}(F) = Attr_{j_0}(F) = \{1, 2, 3, 4, 5, 9, 10, 11\}$

Cet attracteur correspond à l'ensemble gagnant W_0 du joueur j_0. Par exemple, si le pion est initialement placé sur le sommet 5, ce sera à j_0 de le déplacer. Selon la stratégie gagnante pour le joueur j_0 décrite dans la section précédente, celui-ci déplacera le pion vers 3. Le coup suivant sera décidé par le joueur j_1. celui-ci n'aura pas le choix et devra déplacer le pion sur un sommet de F : 1 ou 2. Le joueur j_0 gagne la partie.
L'ensemble $W_1 = \{6, 7, 8\}$ est donc l'ensemble gagnant du joueur j_1. Si le pion est initialement placé sur un sommet de cet ensemble, alors j_1 gagne. Par exemple, le pion placé initialement sur 6 sera déplacé par le joueur j_1 vers 7 (en passant éventuellement par 8 selon la stratégie gagnante pour le joueur j_1 car ni 7 ni 8 ne font partie de l'attracteur). Le joueur j_0 déplacera le pion de 7 vers 6 et la partie consistera en une répétition infinie de coups entre 6, 7 et 8.

Chapitre 4

Cas infini

Résoudre les jeux de sûreté joués sur des *arènes infinies* va nécessiter une étape d'abstraction. En effet, un algorithme linéaire en terme d'arcs et de sommets aurait une complexité infinie sur un graphe de taille infinie. Nous allons donc, dans ce chapitre, introduire et utiliser des notions de la théorie des automates afin de produire un problème de taille finie, et ensuite proposer des algorithmes de programmation logique afin de résoudre celui-ci [2].

Définition 4.0.1. *Une* arène infinie *est une arène $A = (V_0, V_1, E)$ pour laquelle on autorise V_0 et V_1 à être des ensembles infinis.*

4.1 Alphabets, automates et transducteurs

Un *alphabet* Σ est un ensemble fini, non vide, de symboles. Un *mot* de longueur $n \in \mathbb{N}$ sur l'alphabet Σ est une séquence de symboles $u = u_1u_2...u_n$ avec $\forall i \in \mathbb{N}, u_i \in \Sigma$. Le mot vide est noté ϵ.
La *concaténation* de deux mots $u = u_1u_2...u_n$ et $v = v_1v_2...v_m$, avec $n, m \in \mathbb{N}$ est le mot $w = u \cdot v = u_1u_2...u_nv_1v_2...v_m$ de longueur $n + m$. L'ensemble Σ^* contient tous les mots possibles sur l'alphabet Σ.
Un *facteur* $v \in \Sigma^*$ d'un mot $u \in \Sigma^*$ est un mot tel que $\exists w_1, w_2 \in \Sigma^*, w_1vw_2 = u$.
Un *préfixe* $v \in \Sigma^*$ d'un mot $u \in \Sigma^*$ est un mot tel que $\exists w \in \Sigma^*, vw = u$.
Un sous-ensemble $L \subseteq \Sigma^*$ est appelé un langage.

Définition 4.1.1. *L'ensemble des préfixes d'un langage $L \subseteq \Sigma^*$ est l'ensemble $Pref(L) = \{u \in \Sigma^* \mid \exists v \in \Sigma^*, uv \in L\}$.*

Un *automate* fini est un 5-uple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ où
- Q est un ensemble fini, non vide, d'états.
- Σ est l'alphabet lu par l'automate.

— δ est une fonction $Q \times \Sigma \to Q$ appelée *fonction de transition* de l'automate. L'expression $\delta(q_1, u) = q_2$ indique que l'automate transitionne de l'état q_1 à l'état q_2 quand le mot $u \in \Sigma^*$ est lu depuis l'état q_1.
— $q_0 \subseteq Q$ est l'ensemble des états initiaux.
— $F \subseteq Q$ est l'ensemble des états finaux.

Définition 4.1.2 (Automate fini déterministe (AFD))**.** *Un automate $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ est* déterministe *si $\forall q, p_1, p_2 \in Q, a \in \Sigma^*, \delta(q, a) = p_1$ et $\delta(q, a) = p_2$ implique que $p_1 = p_2$.*

Un automate fini qui n'est pas déterministe est appelé *automate fini non déterministe (AFN)*.

Un *parcours* d'un automate par un mot $u = u_1u_2...u_n$, $n \in \mathbb{N}$ est une séquence d'états $q_0q_1...q_n$ telle que $\forall i \in \mathbb{N}, \delta(q_i, u_i) = q_{i+1}$. Un mot u est *accepté* par un automate $\mathcal{A}$ si le parcours de l'automate par ce mot se termine sur un état final $q \in F$. L'ensemble des mots acceptés par un automate forme le langage $\mathcal{L}(\mathcal{A})$ de cet automate. Un langage L est dit *régulier* s'il existe un automate fini $\mathcal{A}$ tel que $\mathcal{L}(\mathcal{A}) = L$.

Enfin, un *transducteur* est un automate fini non déterministe $\mathcal{T} = (Q, \hat{\Sigma}, \delta, q_0, F)$ où $\hat{\Sigma} = (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\})$ avec Σ et Γ deux alphabets. Un parcours d'un transducteur ne se fait pas par un mot mais par une paire de mots (u, v) avec $u \in \Sigma^*$ et $v \in \Gamma^*$ afin de produire une séquence $q_0q_1...q_n$, $n \in \mathbb{N}$ telle que $\forall i \in \mathbb{N}, \delta(q_i, (u_i, v_i)) = q_{i+1}$. Les mots u et v peuvent être de longueur différente car ϵ est inclu aux deux alphabets.
Un transducteur, en acceptant des paires de mots, permet de définir une *relation* R entre deux langages. L'ensemble des paires de mots acceptées par un transducteur forme la relation $\mathcal{R}(\mathcal{T})$ de ce transducteur. Une relation est dite *régulière* s'il existe un transducteur $\mathcal{T}$ tel que $\mathcal{R}(\mathcal{T}) = R$.

Définition 4.1.3 (Image d'une relation [1])**.** *Soit $\mathcal{T}$ un transducteur acceptant la relation R sur $\hat{\Sigma} = (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\})$ et $\mathcal{A}$ un AFN, alors*
$\mathcal{R}(\mathcal{T})(\mathcal{L}(\mathcal{A})) = \{v \in \Gamma^* \mid \exists u \in \Sigma^*, (u, v) \in \mathcal{R}(\mathcal{T})\}$ est l'image du langage $\mathcal{L}(\mathcal{A})$ par la relation R.

Définition 4.1.4 (Inverse d'une relation [1])**.** *Soit $\mathcal{T} = (Q, \hat{\Sigma}, \delta, q_0, F)$ un transducteur acceptant la relation R. On définit l'inverse de R par $R^{-1} = \{(u, v) \mid (v, u) \in R\}$, la relation que l'on obtient en échangeant tous les éléments des paires de R.*

4.2 Exemple de jeu de sûreté infini

Afin d'illustrer le cas infini, nous allons considérer l'arène infinie simple représentée à la figure 4.1. Il s'agit d'un tableau infini, à une dimension, fermé à gauche. Chaque sommet est décrit par le numéro du joueur auquel il appartient, et un nombre $k \in \mathbb{N}$. Il existe un sommet appartenant à chaque joueur pour chaque valeur possible de k et ceux-ci sont arrangés dans l'ordre croissant de cette valeur, en alternant les sommets de j_0 et j_1. L'ensemble F est représenté par les sommets grisés. Il est décrit par une certaine valeur non nulle de k, à partir de laquelle tous les sommets font partie de F (à la figure 4.1, $k = 2$).

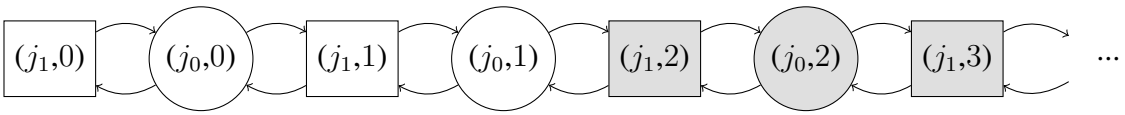


FIGURE 4.1 – Exemple d'arène infinie

Nous appellerons le jeu joué sur cette arène $\mathfrak{G}_k$. Les règles du jeu sont différentes pour cet exemple. Les joueurs bougent le pion chacun à leur tour. Le joueur j_0 ne peut déplacer le pion que vers la droite ou ne pas le déplacer du tout, et le joueur j_1 ne peut le déplacer que vers la gauche (s'il n'est pas arrivé au bout de l'arène) ou ne pas le déplacer. Au début de la partie, le pion est placé sur un sommet appartenant à F, et le joueur j_0 commence. Les objectifs des joueurs restent les mêmes, j_0 veut rester dans F tandis que j_1 essaie d'en sortir. Cet exemple est simple car le joueur j_0 peut gagner, peu importe F, en choisissant de bouger le pion à droite à chaque coup. Il reste néanmoins intéressant à résoudre.

4.3 Jeux de sûreté rationnels

Un jeu de sûreté rationnel est une représentation symbolique d'un jeu de sûreté, basée sur les langages rationnels et les relations rationnelles.

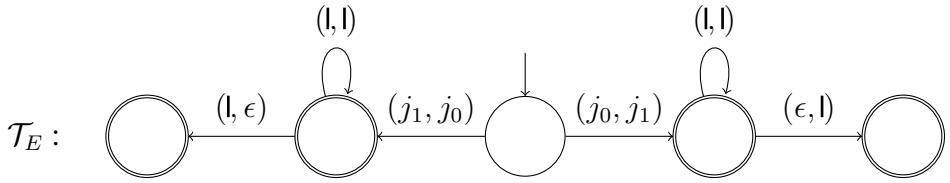
Définition 4.3.1. *Une* arène rationnelle *sur l'alphabet Σ est une arène $A_\Sigma = (V_0, V_1, E)$ où V_0, V_1 sont des langages réguliers et E est une relation régulière.*

Définition 4.3.2. *Un* jeu de sûreté rationnel *sur l'alphabet Σ est un jeu $\mathfrak{G}_\Sigma = (A_\Sigma, I, F)$ où A_Σ est une arène rationnelle sur Σ et $I, F \subseteq \Sigma^*$ sont des langages réguliers.*

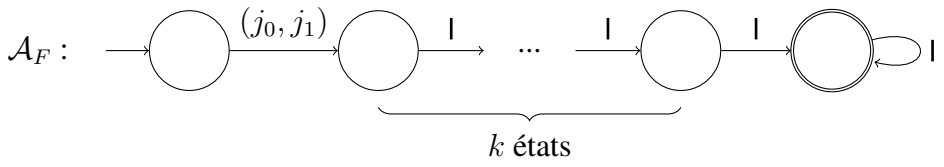
Nous allons maintenant traduire le jeu $\mathfrak{G}_k$ en un jeu de sûreté rationnel. Pour cela, nous devons proposer un encodage qui fait correspondre à chaque arc et chaque

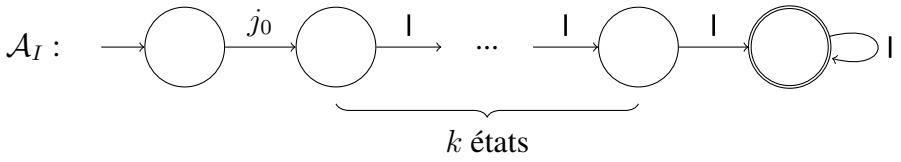
sommet un mot unique d'un alphabet. Pour cela, nous allons considérer l'alphabet $\Sigma = \{j_0, j_1, \mathsf{I}\}$ afin d'associer le sommet $(x, i) \in \{j_0, j_1\} \times \mathbb{N}$ au mot $x\mathsf{I}^i$ où I^i est l'encodage de i en unaire.
Ainsi, les automates suivants représentent les ensembles V_0 et V_1 :

Les arcs (l'ensemble E) sont représentés par le transducteur suivant



Et pour finir, les automates $\mathcal{A}_F$ et $\mathcal{A}_I$ représentent les ensembles F et I.





Avant d'introduire la manière dont on va résoudre ce jeu, nous allons apporter quelques précisions à la définition 2.3.2 d'ensemble gagnant.

Définition 4.3.3. *Soit $\mathfrak{G} = (A, I, F)$ un jeu, avec $A = (V_0, V_1, E)$, l'*ensemble gagnant *pour le joueur j_0 est l'ensemble $W \subseteq V$ qui satisfait :*

- *(1) $I \subseteq W$ (*sommets initiaux*)*
- *(2) $W \subseteq F$ (*sommets sûrs*)*
- *(3) $\forall v \in W \cap V_0$, $Succ(v) \cap W \neq \emptyset$ (*fermeture existentielle*)*
- *(4) $\forall v \in W \cap V_1$, $Succ(v) \subseteq W$ (*fermeture universelle*)*

Où Succ(v) *est la fonction retournant les successeurs du sommet v.*
Cet ensemble n'est pas garanti d'exister pour un jeu de sûreté.

Cette nouvelle définition de l'ensemble gagnant pour un jeu de sûreté nous donne immédiatement une stratégie gagnante pour le joueur j_0. Etant donné que $I \in W_0$, celui-ci n'a qu'à déplacer le pion vers un sommet de W_0 à chaque coup. j_1 n'aura pas le choix que de déplacer son pion vers un sommet dans W_0 lui aussi.

Si un tel ensemble gagnant existe, le joueur j_1 ne peut pas gagner la partie (car $I \in W_0$). Nous allons maintenant décrire la manière de construire un tel ensemble gagnant. Dans le reste de ce rapport, nous allons supposer qu'un tel ensemble existe.

4.4 Apprentissage

Résoudre un jeu de sûreté rationnel consiste à produire un AFD qui acceptera tous les mots représentant les sommets de l'ensemble gagnant W_0. Afin de produire un tel automate, nous allons utiliser le principe d'*apprentissage*. Cet apprentissage à lieu entre un *enseignant*, lequel est en connaissance du jeu de sûreté rationnel que l'on veut résoudre, et un *élève*, cherchant à en apprendre l'AFD décrivant l'ensemble gagnant. On suppose que l'alphabet utilisé est connu à l'avance par les deux partis.

L'apprentissage se déroule à l'intérieur d'une boucle de type *CEGIS* (Counter-example guided inductive synthesis, ou synthèse inductive guidée par le contre-exemple). A chaque itération de cette boucle, l'élève conjecture un AFD $\mathcal{C}$ et l'enseignant vérifie à l'aide de la définition 4.3.3 si $\mathcal{L}(\mathcal{C})$ est un ensemble gagnant. Si c'est le cas, l'enseignant répond "*oui*" et l'apprentissage s'arrête. Sinon, l'enseignant retourne un *contre-exemple* pour une des règles non respectées de la définition 4.3.3 et l'apprentissage se poursuit dans l'itération suivante. Plus formellement, la définition suivante établit le protocole de vérification par l'enseignant :

Définition 4.4.1. *Enseignant pour jeux de sûreté rationnels*
Soit $\mathfrak{G}_\Sigma = (A_\Sigma, I, F)$, un jeu de sûreté rationnel sur Σ, avec $A_\Sigma = (V_0, V_1, E)$. Soit un AFD $\mathcal{C}$. Un enseignant pour $\mathfrak{G}_\Sigma$ confronté à $\mathcal{C}$ va effectuer les vérifications suivantes :

- *(1) Si $I \not\subseteq \mathcal{L}(\mathcal{C})$, alors l'enseignant retourne un* contre-exemple positif *$u \in I \setminus \mathcal{L}(\mathcal{C})$. Sinon,*
- *(2) Si $\mathcal{L}(\mathcal{C}) \not\subseteq F$, alors l'enseignant retourne un* contre-exemple négatif *$u \in \mathcal{L}(\mathcal{C}) \setminus F$. Sinon,*

- *(3) Si* $\exists u \in \mathcal{L}(\mathcal{C}) \cap V_0$, $Succ(u) \cap \mathcal{L}(\mathcal{C}) = \emptyset$ *alors l'enseignant choisit ce* u *et retourne un* contre-exemple d'implication existentielle $(u, \mathcal{A}) \in \Sigma^* \times AFN_\Sigma$ *où* $\mathcal{L}(\mathcal{A}) = Succ(u)$. *Sinon,*
- *(4) Si* $\exists u \in \mathcal{L}(\mathcal{C}) \cap V_1$, $Succ(u) \nsubseteq \mathcal{L}(\mathcal{C})$, *alors l'enseignant choisit ce* u *et retourne un* contre-exemple d'implication universelle $(u, \mathcal{A}) \in \Sigma^* \times AFN_\Sigma$ *où* $\mathcal{L}(\mathcal{A}) = Succ(u)$.

Où

- AFN_Σ *est l'ensemble des automates finis non-déterministes sur* Σ.
- *Un* contre-exemple positif *est un mot* u *qui devrait être accepté par* $\mathcal{C}$*, mais qui a été rejeté.*
- *Un* contre-exemple négatif *est un mot* u *qui devrait être rejeté par* $\mathcal{C}$*, mais qui a été accepté.*
- *Un* contre-exemple d'implication existentielle $(u, \mathcal{A})$ *signifie que si* $\mathcal{C}$ *accepte* u*, alors au moins un mot* $v \in \mathcal{L}(\mathcal{A})$ *doit, lui aussi, être accepté, et cela n'a pas été respecté.*
- *Un* contre-exemple d'implication universelle $(u, \mathcal{A})$ *signifie que si* $\mathcal{C}$ *accepte* u*, alors tous les mots* $v \in \mathcal{L}(\mathcal{A})$ *doivent, eux aussi, être acceptés, et cela n'a pas été respecté.*

*Si les 4 vérifications ne retournent pas de contre-exemples, alors l'enseignant répond "*oui*".*
L'ordre des vérifications n'a pas d'importance.

Afin de se souvenir de ces contre-exemples, l'élève les enregistre dans un *échantillon*, lequel est un 4-uple $S = (Pos, Neg, Ex, Uni)$ composé de 4 ensembles finis. L'ensemble Pos (resp. Neg) $\subset \Sigma^*$ contient les contre-exemples positifs (resp. négatifs), et l'ensemble Ex (resp. Uni) $\subset (\Sigma^* \times AFN_\Sigma)$ contient les contre-exemples d'implication existentielle (resp. universelle).

Définition 4.4.2. *Un échantillon* $S = (Pos, Neg, Ex, Uni)$ *est* sans contradictions *si*

1. $Pos \cap Neg = \emptyset$.
   Si Pos *et* Neg *ne sont pas disjoints, alors les mots* $u \in Pos \cap Neg$ *devraient à la fois être acceptés et rejetés par l'ensemble gagnant.*
2. *La fermeture transitive (alternée) des contre-exemples d'implication de* Ex *et* Uni *ne contient pas de paire de mots* (u, v) *avec* u *antécédent d'un contre-exemple de* Uni *telle que* $(u \in Pos) \wedge (v \in Neg)$.
   Si un telle paire existe, alors à nouveau, le mot v *devrait à la fois être accepté et rejeté par l'ensemble gagnant.*

Ainsi, l'élève devra s'assurer que les AFD qu'il soumet à l'enseignant sont *cohérents* avec l'échantillon S.

Définition 4.4.3. *Soit $\mathcal{C}$ un AFD, $S = (Pos, Neg, Ex, Uni)$ un échantillon sans contradictions. On dit que $\mathcal{C}$ est* cohérent *avec S si*

- $Pos \subseteq \mathcal{L}(\mathcal{C})$
- $Neg \cap \mathcal{L}(\mathcal{C}) = \emptyset$
- $\forall (u, \mathcal{A}) \in Ex, u \in \mathcal{L}(\mathcal{C}) \implies \mathcal{L}(\mathcal{C}) \cap \mathcal{L}(\mathcal{A}) \neq \emptyset$
- $\forall (u, \mathcal{A}) \in Uni, u \in \mathcal{L}(\mathcal{C}) \implies \mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{C})$

Pour finir, si le contre-exemple retourné par l'enseignant engendre une contradiction dans S, alors l'enseignant peut en conclure que le jeu de sûreté est toujours gagné par le joueur j_1 (i.e. l'ensemble gagnant pour le joueur j_0 est $W_0 = \emptyset$).

La section suivante va maintenant pouvoir décrire la manière dont l'élève peut conjecturer un AFD cohérent avec un échantillon S.

4.5 Résolution par la programmation logique

Afin de construire un AFD cohérent avec un échantillon S donné, une nouvelle étape de symbolisation va être nécessaire. Cette fois-ci, il s'agit de traduire le problème rationnel en une série de formules boléennes propositionnelles et de résoudre les problèmes de satisfaisabilité (*problème SAT*) résultants. Il sera ensuite possible de traduire les *modèles* (assignations de variables boléennes rendant la formule vraie) de ce problème SAT en ensemble gagnant pour le jeu de sûreté.

Définition 4.5.1. *Elève pour les jeux de sûreté rationnels*
Soit $\mathfrak{G}_\Sigma = (A_\Sigma, I, F)$, un jeu de sûreté rationnel sur Σ, avec $A_\Sigma = (V_0, V_1, E)$ et $S = (Pos, Neg, Uni, Ex)$ un échantillon. Afin d'apprendre un AFD $\mathcal{C}$ à soumettre à l'enseignant, l'élève pour $\mathfrak{G}_\Sigma$ va créer une série de formules boléennes ϕ_n^S pour une valeur incrémentale de $n \in \mathbb{N}_0$ et telle que :

- *La formule ϕ_n^S est satisfaisable si et seulement si il existe un AFD qui possède n sommets et qui est cohérent avec S.*
- *Un modèle $\mathfrak{M}$ de ϕ_n^S contient assez d'information pour construire un AFD $\mathcal{A}_\mathfrak{M}$ à n sommets cohérent avec S.*

Autrement formulé, l'objectif de ϕ_n^S est d'encoder un AFD $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ où $|Q| = n$ et cohérent avec S.
L'élève soumettra à l'enseignant un AFD $\mathcal{C}$ minimal, c.-à-d. construit à partir du modèle $\mathfrak{M} \models \phi_n^S$ pour la plus petite valeur de n possible.

La manière dont l'élève va construire cet AFD est la suivante. Etant donné n, on peut à l'avance fixer l'ensemble des états de cet AFD à $Q = \{0, 1, ..., n-1\}$ et le sommet de départ $q_0 = 0$. Ainsi, la formule ϕ_n^S n'a plus qu'à en encoder les transitions δ et les états finaux F (pour rappel, Σ est connu à l'avance par l'élève). Afin

de représenter ces deux ensembles, nous allons introduire les variables boléennes f_q et $d_{p,a,q}$ avec $a \in \Sigma$ et $p, q \in Q$. La variable $f_q = vrai$ signifie que q est un état final de $\mathcal{A}$ et $d_{p,a,q} = vrai$ signifie que la transition $\delta(p, a)$ existe et transitionne vers q. De plus, vu que l'on cherche à encoder un AFD, et non un AFN, il faut imposer des conditions à nos variables, afin de respecter la définition 4.1.2. Ces conditions s'obtiennent par la formule

$$\bigwedge_{p \in Q} \bigwedge_{a \in \Sigma} \bigwedge_{q,q' \in Q, q \neq q'} \neg(d_{p,a,q} \wedge d_{p,a,q'}) \tag{4.1}$$

qui assure qu'au plus un état q peut être atteint depuis un état p en lisant le mot a. A cela, nous allons ajouter la formule

$$\bigwedge_{p \in Q} \bigwedge_{a \in \Sigma} \bigvee_{q \in Q} d_{p,a,q} \tag{4.2}$$

qui indique que la fonction de transition est totale, autrement dit que $\forall p \in Q, a \in \Sigma, \exists q \in Q, \delta(p, a) = q$. Bien que cela ne soit pas nécessaire pour encoder une fonction de transition déterministe, cela sera intéressant pour les formules que nous allons définir par la suite.
Ainsi, il est possible pour l'élève de construire un AFD $\mathcal{A}_{\mathfrak{M}} = (Q, \Sigma, q_0, \delta, F)$ à partir d'un modèle $\mathfrak{M}$ satisfaisant $\phi_n^{AFD} = (4.1) \wedge (4.2)$.

Définition 4.5.2. *Soit $\mathfrak{M}$ un modèle satisfaisant $\phi_n = (4.1) \wedge (4.2)$, alors on peut construire un AFD $\mathcal{A}_{\mathfrak{M}} = (Q, \Sigma, \delta, q_0, F)$ avec Q, q_0 et Σ fixés, $\delta(p, a) = q$ pour tout $\mathfrak{M}(d_{p,a,q}) =$ vrai, et $F = \{q \in Q \mid \mathfrak{M}(f_q) =$ vrai $\}$*
Où $\mathfrak{M}(x)$ correspond à la valeur de x dans le modèle $\mathfrak{M}$.

Il reste maintenant à rendre cet AFD cohérent avec S. Pour cela, il est nécessaire que $\mathcal{A}_{\mathfrak{M}}$ accepte ou rejette tous les mots de S, définis par l'ensemble $W = Pos \cup Neg \cup Ante(Ex) \cup Ante(Uni)$ (où, pour un contre-exemple d'implication $i = (u, \mathcal{A})$, on a $Ante(i) = u$). Autrement dit, il faut qu'il existe un parcours dans $\mathcal{A}_{\mathfrak{M}}$ pour chaque préfixe des mots de W. Nous allons donc introduire une nouvelle variable boléenne $x_{u,q}$ pour laquelle $x_{u,q} = vrai$ signifie que $\mathcal{A}_{\mathfrak{M}}$ atteint l'état $q \in Q$ en lisant le mot u et utiliser cette variable pour décrire le parcours de $\mathcal{A}_{\mathfrak{M}}$ par les mots de l'ensemble $Pref(W)$.

Nous commencons par introduire la formule

$$x_{\varepsilon,q_0} \tag{4.3}$$

laquelle indique que n'importe quel parcours de $\mathcal{A}_{\mathfrak{M}}$ commence par l'état q_0.
Ensuite, la formule

$$\bigwedge_{u \in Pref(W)} \bigwedge_{q \neq q' \in Q} \neg(x_{u,q} \wedge x_{u,q'}) \tag{4.4}$$

assure que pour chaque mot u de l'ensemble des préfixes des mots de l'échantillon S, il existe au plus un état q tel que $x_{u,q} = vrai$. D'ailleurs, cet état q est garanti d'exister par la formule 4.2 introduite précédemment.
Pour finir, la formule

$$\bigwedge_{ua \in Pref(W)} \bigwedge_{p,q \in Q} (x_{u,p} \wedge d_{p,a,q}) \rightarrow x_{ua,q} \tag{4.5}$$

décrit la manière dont un parcours de l'automate $\mathcal{A}_{\mathfrak{M}}$ par un mot $ua \in Pref(W)$ fonctionne. Si un état p est atteint en lisant le mot u, et qu'une transition $\delta(p, a) = q$ existe (la variable $d_{p,a,q} = vrai$) alors l'état q sera atteint en lisant le mot ua et donc $x_{ua,q} = vrai$.

La conjonction de ces trois formules $\phi_n^W = (4.3) \wedge (4.4) \wedge (4.5)$ nous permet ainsi de produire un AFD $\mathcal{A}_{\mathfrak{M}}$ qui permettra d'accepter ou de rejetter (i.e. d'être parcouru par) tous les mots de W.

Pour rendre un AFD $\mathcal{A}_{\mathfrak{M}}$ construit à partir d'un modèle $\mathfrak{M}$ de $\phi_n = \phi_n^{AFD} \wedge \phi_n^W$ cohérent avec S, il reste maintenant à définir des formules boléennes ϕ_n^{Pos}, ϕ_n^{Neg}, ϕ_n^{Ex} et ϕ_n^{Uni} qui assureront le respect des conditions de la définition 4.4.3.

Premièrement, la formule suivante pour ϕ_n^{Pos}

$$\bigwedge_{u \in Pos} \bigwedge_{q \in Q} x_{u,q} \rightarrow f_q \tag{4.6}$$

permet de s'assurer que les mots de Pos sont acceptés par $\mathcal{A}_{\mathfrak{M}}$. En effet, si $x_{u,q} = vrai$, alors q est l'état atteint lors de la lecture du mot u par $\mathcal{A}_{\mathfrak{M}}$. Par cette formule, une valeur $x_{u,q} = vrai$ impliquera $f_q = vrai$ et donc q sera un état final, ce qui permettra d'accepter u. Nous avons donc bien que si $\mathfrak{M} \models \phi_n^{Pos} \wedge \phi_n^W \wedge \phi_n^{AFD}$, alors $Pos \subseteq \mathcal{L}(\mathcal{A}_{\mathfrak{M}})$

A l'inverse, la formule suivante pour ϕ_n^{Neg}

$$\bigwedge_{u \in Neg} \bigwedge_{q \in Q} x_{u,q} \rightarrow \neg f_q \tag{4.7}$$

assurera, par un raisonnement similaire, que les mots de Neg seront rejetés par $\mathcal{A}_{\mathfrak{M}}$, et donc que si $\mathfrak{M} \models \phi_n^{Neg} \wedge \phi_n^W \wedge \phi_n^{AFD}$, alors $\mathcal{L}(\mathcal{A}_{\mathfrak{M}}) \cap Neg = \emptyset$.

Ensuite, afin que la formule pour ϕ_n^{Uni} assure que pour tout contre-exemple d'implication universelle $i_{Uni} = (u, \mathcal{A}) \in Uni$, avec $\mathcal{A} = (Q^{\mathcal{A}}, \Sigma, q_0^{\mathcal{A}}, \delta^{\mathcal{A}}, F^{\mathcal{A}})$, si

$\mathcal{A}_{\mathfrak{M}}$ accepte u, alors il doit aussi accepter tous les $v \in \mathcal{L}(\mathcal{A})$, nous allons définir des variables boléennes $y_{q,q^{\mathcal{A}}}$ telles que $y_{q,q^{\mathcal{A}}} = vrai$ indique qu'il existe un mot $v \in \Sigma^*$ pour lequel un parcours de $\mathcal{A}_{\mathfrak{M}}$ par v se termine sur l'état $q \in Q$ et un parcours de $\mathcal{A}$ par ce même v se termine sur l'état $q^{\mathcal{A}} \in Q^{\mathcal{A}}$. Autrement formulé, ces variables vont nous aider à déterminer les paires d'états atteintes par un parcours en parallèle de $\mathcal{A}_{\mathfrak{M}}$ et $\mathcal{A}$ par v.
Soit $i_{Uni} = (u, \mathcal{A}) \in Uni$, un contre-exemple d'implication universelle, avec $\mathcal{A} = (Q^{\mathcal{A}}, \Sigma, q_0^{\mathcal{A}}, \delta^{\mathcal{A}}, F^{\mathcal{A}})$, cette définition est donnée par

$$y_{q_0,q_0^{\mathcal{A}}} \tag{4.8}$$

qui indique que tout parcours en parallèle de $\mathcal{A}_{\mathfrak{M}}$ et $\mathcal{A}$ commence sur la paire d'état $(q_0, q_0^{\mathcal{A}})$, et

$$\bigwedge_{p,q \in Q} \bigwedge_{(p^{\mathcal{A}},a,q^{\mathcal{A}}) \in \delta^{\mathcal{A}}} (y_{p,p^{\mathcal{A}}} \wedge d_{p,a,q}) \rightarrow y_{q,q^{\mathcal{A}}} \tag{4.9}$$

qui, de manière analogue à la formule 4.5, indique que si une paire d'états $(p, p^{\mathcal{A}}) \in Q \times Q^{\mathcal{A}}$ est atteinte par le parcours en parallèle de $\mathcal{A}_{\mathfrak{M}}$ et $\mathcal{A}$ par un mot de Σ^*, alors pour toutes transitions $\delta^{\mathcal{A}}(p^{\mathcal{A}}, a) = q^{\mathcal{A}}$, s'il y a une transition $\delta(p, a) = q$ dans $\mathcal{A}_{\mathfrak{M}}$ (ce qui est toujours le cas par 4.2 pour $a \in \Sigma \setminus \epsilon$), alors on peut transitionner via a vers la paire d'états $(q, q^{\mathcal{A}}) \in Q \times Q^{\mathcal{A}}$ et donc $y_{q,q^{\mathcal{A}}} = vrai$.

Grâce à cette définition de $y_{q,q^{\mathcal{A}}}$, on peut assurer, par la formule

$$(\bigvee_{q \in Q} x_{u,q} \wedge f_q) \rightarrow (\bigwedge_{q \in Q} \bigwedge_{q^{\mathcal{A}} \in F^{\mathcal{A}}} y_{q,q^{\mathcal{A}}} \rightarrow f_q) \tag{4.10}$$

que si un état final est atteint dans $\mathcal{A}_{\mathfrak{M}}$ en lisant le mot u, alors le parcours de $\mathcal{A}_{\mathfrak{M}}$ pour chaque mot de $\mathcal{L}(\mathcal{A})$ mène aussi à un état final de $\mathcal{A}_{\mathfrak{M}}$.

En construisant une formule $\phi_{n,i}^{Uni} = (4.8) \wedge (4.9) \wedge (4.10)$ pour chaque contre-exemple d'implication universelle $i \in Uni$, on obtient la définition de la formule $\phi_n^{Uni} = \bigwedge_{i \in (Uni)} \phi_{n,i}^{Uni}$.
Par la construction que l'on vient de décrire, on a bien que si $\mathfrak{M} \models \phi_n^{Uni} \wedge \phi_n^{W} \wedge \phi_n^{AFD}$, alors $\forall (u, \mathcal{A}) \in Uni, u \in \mathcal{L}(\mathcal{A}_{\mathfrak{M}}) \implies \mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}_{\mathfrak{M}})$

Cependant, il est important de noter que les variables $y_{q,q^{\mathcal{A}}}$ ne décrivent pas exactement le parcours en parallèle de $\mathcal{A}_{\mathfrak{M}}$ et $\mathcal{A}$ par les mots de $\mathcal{L}(\mathcal{A})$. En effet, étant donné que $\mathcal{A}$ est non-déterministe, une transition $\delta^{\mathcal{A}}(p^{\mathcal{A}}, \epsilon) = q^{\mathcal{A}}$ pourrait exister, ce qui rendrait la prémisse de la formule 4.9 fausse car $\mathcal{A}_{\mathfrak{M}}$ est déterministe et donc ne possède pas de transitions pour ϵ. L'implication aura donc pour conséquence

$y_{q,q^{\mathcal{A}}} = vrai$ sans qu'un mot v existe tel que le parcours en parallèle de $\mathcal{A}_{\mathfrak{M}}$ et $\mathcal{A}$ par v se termine sur la paire d'états $(q, q^{\mathcal{A}}) \in Q \times Q^{\mathcal{A}}$.
La formule ϕ_n^{Uni} reste cependant correcte. En effet, dans l'objectif d'accepter tous les mots $v \in \mathcal{L}(\mathcal{A})$, il n'est pas nécessaire de savoir s'ils existent. Par contre, afin que la formule ϕ_n^{Ex} que nous allons construire par la suite assure bien que, pour un contre-exemple d'implication existentielle $i_{Ex} = (u, \mathcal{A}) \in Ex$, si $u \in \mathcal{L}(\mathcal{A}_{\mathfrak{M}})$, alors il existe au moins un $v \in \mathcal{L}(\mathcal{A})$ tel que $v \in \mathcal{L}(\mathcal{A}_{\mathfrak{M}})$, il n'est plus suffisant de se baser sur les variables $y_{q,q^{\mathcal{A}}}$ car elle n'assurent pas l'existence d'un tel mot. Nous allons devoir procéder de manière plus robuste.

Soit $i_{Ex} = (u, \mathcal{A}) \in Ex$ un contre-exemple d'implication existentielle, avec $\mathcal{A} = (Q^{\mathcal{A}}, \Sigma, q_0^{\mathcal{A}}, \delta^{\mathcal{A}}, F^{\mathcal{A}})$, nous allons maintenant introduire une dernière variable boléenne $z_{q,q^{\mathcal{A}},l}$ avec $q \in Q$, $q^{\mathcal{A}} \in Q^{\mathcal{A}}$ et $l \in \{0, \dots, k\}$, $k \in \mathbb{N}$ pour laquelle $z_{q,q^{\mathcal{A}},l} = vrai$ indique qu'il existe un mot $v \in \Sigma^*$ de longueur $|v| = l$ tel que le parcours en parallèle de $\mathcal{A}_{\mathfrak{M}}$ et $\mathcal{A}$ par v se termine sur la paire d'états $(q, q^{\mathcal{A}}) \in Q \times Q^{\mathcal{A}}$.

Afin que le nombre de variables nécessaires soit fini, il faut déterminer une valeur maximale pour l. Il faut donc trouver la plus petite longueur maximale de mot de Σ^* pour laquelle $z_{q,q^{\mathcal{A}},l}$ permet de décrire tous les parcours en parallèle de $\mathcal{A}_{\mathfrak{M}}$ et $\mathcal{A}$. Nous allons déterminer cette valeur sur base d'une simple observation : si, lors du parcours en parallèle de $\mathcal{A}_{\mathfrak{M}}$ et $\mathcal{A}$ par un mot $v \in \Sigma$ un état $(q, q^{\mathcal{A}}) \in Q \times Q^{\mathcal{A}}$ est atteint deux fois, alors le facteur $w \in \Sigma^*$ de v entre cette répétition peut être retiré de v sans que $z_{q,q^{\mathcal{A}},l}$ ne perde d'information. Ainsi, la valeur maximale pour l est la longueur maximale de mot telle que, lors du parcours en parallèle de $\mathcal{A}_{\mathfrak{M}}$ et $\mathcal{A}$ par ce mot, chaque paire d'état $(q, q^{\mathcal{A}}) \in Q \times Q^{\mathcal{A}}$ est atteinte au plus une fois. Cette valeur sera donc donnée par $k = |Q||Q^{\mathcal{A}}| - 1 = n|Q^{\mathcal{A}}| - 1$

Maintenant que nous avons réussi à limiter le nombre de ces variables, passons à leur définition.
Pour commencer, la formule

$$z_{q_0,q_0^{\mathcal{A}},0} \tag{4.11}$$

permet d'indiquer, de manière similaire aux formules 4.3 et 4.8, qu'un parcours en parallèle $\mathcal{A}_{\mathfrak{M}}$ et $\mathcal{A}$ commencera sur la paire d'états $(q_0, q_0^{\mathcal{A}}) \in Q \times Q^{\mathcal{A}}$.
Ajouté à cela, la formule

$$\bigwedge_{(q,q^{\mathcal{A}}) \in Q \times Q^{\mathcal{A}} \setminus \{(q_0,q_0^{\mathcal{A}})\}} \neg z_{q,q^{\mathcal{A}},0} \tag{4.12}$$

assure que toutes les autres variables $z_{q,q^{\mathcal{A}},0} = faux$ car le seul parcours valide par le mot ϵ (i.e. par un mot de longueur 0) selon le sens que l'on veut donner à

notre variable $z_{q,q^{\mathcal{A}},l}$ est celui qui amène à la paire d'états initiaux comme décrit par 4.11.
Ensuite, la formule

$$\bigwedge_{p,q\in Q} \bigwedge_{(p^{\mathcal{A}},a,q^{\mathcal{A}})\in\delta^{\mathcal{A}}} \bigwedge_{l\in\{0,\dots,k-1\}} (z_{p,p^{\mathcal{A}},l} \wedge d_{p,a,q}) \rightarrow z_{q,q^{\mathcal{A}},l+1} \tag{4.13}$$

décrit, de manière analogue à 4.4 et 4.9, que si la paire d'états $(p, p^{\mathcal{A}}) \in Q \times Q^{\mathcal{A}}$ est atteinte lors du parcours en parallèle de $\mathcal{A}_{\mathfrak{M}}$ et $\mathcal{A}$ par un mot de longueur l, et qu'il existe les transitions $\delta(p, a) = q$ et $\delta^{\mathcal{A}}(p^{\mathcal{A}}, a) = q^{\mathcal{A}}$ avec $q \in Q$ et $q^{\mathcal{A}} \in Q^{\mathcal{A}}$, alors on atteint la paire d'états $(q, q^{\mathcal{A}})$ par le parcours d'un mot de longueur $l + 1$ et donc $z_{q,q^{\mathcal{A}},l+1} = vrai$.
Pour finir, afin d'assurer l'existence de ce mot de longueur $l + 1$, la formule

$$(\bigwedge_{q\in Q} \bigwedge_{q^{\mathcal{A}}\in Q^{\mathcal{A}}} \bigwedge_{l\in\{0,\dots,k-1\}} z_{q,q^{\mathcal{A}},l+1}) \rightarrow (\bigvee_{p\in Q} \bigvee_{(p^{\mathcal{A}},a,q^{\mathcal{A}})\in\delta^{\mathcal{A}}} d_{p,a,q} \wedge z_{p,p^{\mathcal{A}},l}) \tag{4.14}$$

empêche $z_{q,q^{\mathcal{A}},l+1} = vrai$ s'il n'y a pas de paire d'états $(p, p^{\mathcal{A}})$ atteinte par un mot de longueur l tel que $z_{p,p^{\mathcal{A}},l} = vrai$. Cela empêche la formule 4.13 de rendre $z_{q,q^{\mathcal{A}},l+1} = vrai$ s'il est atteint par une transition $\delta^{\mathcal{A}}(p^{\mathcal{A}}, \epsilon, q^{\mathcal{A}})$ car la longueur du mot lu n'aura pas été incrémentée.

Grâce à cette définition de $z_{q,q^{\mathcal{A}},l}$, on peut assurer, par la formule

$$(\bigvee_{q\in Q} x_{u,q} \wedge f_q) \rightarrow (\bigvee_{q\in Q} \bigvee_{q^{\mathcal{A}}\in Q^{\mathcal{A}}} \bigvee_{l\in\{0,\dots,k\}} z_{q,q^{\mathcal{A}},l} \wedge f_q) \tag{4.15}$$

que, si un mot u est accepté par $\mathcal{A}_{\mathfrak{M}}$, alors au moins un des mots $v \in \mathcal{L}(\mathcal{A})$ menant à un état final dans $\mathcal{A}$, mènera à un état final dans $\mathcal{A}_{\mathfrak{M}}$ et sera donc accepté par ce dernier.

En construisant une formule $\phi^{Ex}_{n,i} = (4.11) \wedge (4.12) \wedge (4.13) \wedge (4.14) \wedge (4.15)$ pour chaque contre-exemple d'implication existentielle $i \in Ex$, on obtient la définition de la formule $\phi^{Ex}_n = \bigwedge_{i\in(Ex)} \phi^{Ex}_{n,i}$.
Par la construction de cette formule, on a bien que si $\mathfrak{M} \models \phi^{Ex}_n \wedge \phi^{W}_n \wedge \phi^{AFD}_n$, alors $\forall(u, \mathcal{A}) \in Ex, u \in \mathcal{L}(\mathcal{A}) \implies \mathcal{L}(\mathcal{A}_{\mathfrak{M}}) \cap \mathcal{L}(\mathcal{A}) \neq \emptyset$

Ainsi, nous obtenons une formule $\phi^{S}_n = \phi^{AFD}_n \wedge \phi^{W}_n \wedge \phi^{Pos}_n \wedge \phi^{Neg}_n \wedge \phi^{Ex}_n \wedge \phi^{Uni}_n$ dont un modèle $\mathfrak{M}$ contiendra assez d'information pour construire un AFD $\mathcal{A}_{\mathfrak{M}}$ à n états et cohérent avec l'échantillon S.

Tous les outils nécessaires à l'apprentissage étant maintenant définis, la section suivante va détailler l'échange entre l'enseignant et l'élève menant à la construction de l'ensemble gagnant d'un problème de sûreté.

4.6 Implémentation

Nous allons maintenant décrire les structures de données et les algorithmes qui vont permettre l'apprentissage d'un ensemble gagnant pour les jeux de sûreté. Cette description débutera par des algorithmes généraux, peu détaillés, lesquels seront au fur et à mesure décomposés afin d'être expliqués plus en profondeur. Ensuite nous calculerons la complexité en temps de ces algorithmes. Cette section se terminera enfin sur un exemple du déroulement de la résolution d'un jeu de sûreté rationnel.

4.6.1 Structures de données

Afin de stocker les mots de $W = Pos \cup Neg \cup Ante(Ex) \cup Ante(Uni)$ pour un échantillon $S = (Pos, Neg, Uni, Ex)$, nous allons utiliser un *arbre des préfixes*. La définiton de cette structure de données est la suivante :

Définition 4.6.1. *Arbre des préfixes*
*Soit Σ un alphabet, W une liste de mots de Σ^*. L'*arbre des préfixes *de W est la structure de données pour laquelle :*

- *Un* noeud *de cet arbre contient un mot $u \in Pref(W)$ et une liste* succ *de noeuds de taille maximale $|\Sigma|$ qui sont ses successeurs.*
- *La racine de l'arbre contient le mot vide ϵ. Un arbre ne contenant que sa racine est appelé l'*arbre vide.
- *Pour chaque successeur $n \in \{0, ..., |\Sigma| - 1\}$ de la liste des successeurs d'un noeud de l'arbre, lequel contient un mot $u \in Pref(W)$, le mot contenu par le n^{eme} successeur est le mot $ua \in Pref(W)$ tel que a est le n^{eme} caractère de Σ. Un successeur n'est pas garanti d'exister pour toute valeur de n pour un noeud.*

Cette structure de données est utile pour l'apprentissage car elle permet, en plus de stocker efficacement tous les mots de $Pref(W)$, d'établir les liens entre tous ces mots. Cela facilitera la création des variables boléennes $x_{u,q}$ décrites à la section 4.4.
Dans le cadre de l'apprentissage, nous allons augmenter la définition de cet arbre en ajoutant, à chaque noeud, un entier indiquant si le mot contenu dans ce noeud doit être accepté ou rejeté (ou si ce n'est pas défini) par l'AFD que l'élève va chercher à conjecturer. Ainsi, un entier $accept \in \{-1, 0, 1\}$ est contenu dans chaque noeud tel que

- $accept = -1$ signifie que le préfixe doit être rejeté par l'AFD.
- $accept = 0$ indique l'absence de décision.
- $accept = 1$ signifie que le préfixe doit être accepté par l'AFD.

Une fonction $add(u, a)$ permet d'ajouter un mot $u \in \Sigma^*$ à cet arbre et de définir sa valeur pour $accept$. Si un noeud contenant ce mot existe déjà, il est retourné (si ce noeud possède une valeur $accept \neq 0$, alors elle n'est pas changée. Sinon, elle sera mise à $accept = a$), sinon, un parcours de l'arbre jusqu'à un noeud contenant le mot u sera créé (i.e. le mot et tous ses préfixes absents de l'arbre y seront ajoutés) et le noeud terminal de ce parcours sera retourné (la valeur $accept = a$ sera ajoutée au noeud terminal tandis qu'elle sera initialisée à $accept = 0$ pour tous les autres préfixes créés).
Pour finir, la *taille* $|arbre|$ d'un arbre des préfixes est le nombre de noeuds qu'il contient et chaque noeud aura un identifiant unique $id \in \{0, ... |arbre| - 1\}$.

4.6.2 Algorithmes

Soit un jeu de sûreté rationnel $\mathfrak{G}_\Sigma = (A_\Sigma, \mathcal{A}_I, \mathcal{A}_F)$ sur Σ, avec $A_\Sigma = (\mathcal{A}_{V_0}, \mathcal{A}_{V_1}, \mathcal{T}_E)$ que nous allons fixer pour le reste de cette section. On suppose l'alphabet Σ défini globalement. L'algorithme 2 décrit, de manière générale, l'échange entre l'élève et l'enseignant pour $\mathfrak{G}_\Sigma$.

Algorithm 2 Apprentissage

Entrées $\mathfrak{G}_\Sigma$: Un jeu de sûreté rationnel sur Σ
teacher : Un enseignant pour $\mathfrak{G}_\Sigma$
learner : Un élève pour $\mathfrak{G}_\Sigma$
Sortie Un AFD décrivant W_0 l'ensemble gagnant de $\mathfrak{G}_\Sigma$ pour le joueur j_0

$S \leftarrow (\emptyset, \emptyset, \emptyset, \emptyset)$ ▷ Echantillon
$PrefixTree \leftarrow$ arbre des préfixes vide. ▷ 4.6.1
repeat
  $\mathcal{A}_S \leftarrow learner.conjecture(S, PrefixTree)$ ▷ 4.5.1
  $counterexample \leftarrow teacher.check(\mathfrak{G}_\Sigma, \mathcal{A}_S)$ ▷ 4.4.1
  if $counterexample \neq \emptyset$ **then**
    Ajouter $counterexample$ dans S
    if $counterexample$ est un contre-exemple positif u **then**
      $PrefixTree.add(u, 1)$
    else if $counterexample$ est un contre-exemple négatif u **then**
      $PrefixTree.add(u, -1)$
    else
      $PrefixTree.add(Ante(counterexample), 0)$
    end if
  end if
until $counterexample = \emptyset$
return $\mathcal{A}_S$

Avant de démarrer l'apprentissage, on initialise l'échantillon $S = (Pos, Neg, Uni, Ex)$ et l'arbre des préfixes $PrefixTree$ vides. C'est ensuite dans la boucle principale de l'algorithme que se déroulera toute l'étape d'apprentissage. Il s'agit donc de la boucle de type $CEGIS$ décrite dans la section 4.4. Chaque itération de cette boucle commence par l'élève conjecturant un AFD $\mathcal{A}_S$ cohérent avec S. La méthode $learner.conjecture$ est décrite par l'algorithme 3.

Algorithm 3 learner.conjecture

Entrées **S** : L'échantillon (sans contradictions) actuel pour l'apprentissage.
PrefixTree : L'arbre des préfixes actuel pour l'apprentissage.
Sortie Un AFD minimal cohérent avec S.

1: $n \leftarrow 0$
2: **repeat**
3: $\quad n \leftarrow n + 1$
4: $\quad \phi_n^S \leftarrow learner.constructSAT(S, PrefixTree, n)$
5: $\quad \mathfrak{M} \leftarrow learner.solveSAT(\phi_n^S)$ ▷ $\mathfrak{M}$ n'existe peut-être pas
6: **until** $\mathfrak{M} \models \phi_n^S$
7: $\mathcal{A}_{\mathfrak{M}} \leftarrow learner.translatemodel(\mathfrak{M}, n)$
8: **return** $\mathcal{A}_{\mathfrak{M}}$

Afin de conjecturer un AFD minimal cohérent avec S, l'élève va, par la méthode $learner.constructSAT$, construire un problème SAT correspondant à la formule ϕ_n^S décrite à la section 4.5 pour une valeur de n démarrant à $n = 1$ (Elle est initialisée à 0 mais est immédiatement incrémentée dans la boucle) pour ensuite essayer de résoudre ce problème à l'aide de la méthode $learner.solveSAT$. En pratique, cette méthode va correspondre à l'utilisation d'un *solveur SAT*, un outil servant à résoudre les problèmes SAT. Si aucun modèle n'est retourné par le solveur SAT pour la valeur actuelle de n, on passe à l'itération suivante avec une valeur de n incrémentée.
Dès qu'un modèle $\mathfrak{M}$ est trouvé, on sort de la boucle.
La fonction $learner.translatemodel$, décrite par l'algorithme 5, va construire l'AFD minimal $\mathcal{A}_{\mathfrak{M}}$ correspondant à $\mathfrak{M}$, lequel sera la valeur de retour.

La manière dont l'élève va construire le problème SAT est décrite dans l'algorithme 4.

Algorithm 4 leaner.constructSAT

Entrées	**S** :	L'échantillon (sans contradictions) actuel pour l'apprentissage.
	prefixTree :	L'arbre des préfixes actuel pour l'apprentissage.
	n :	La taille (nombre d'états de l'AFD correspondant) du problème SAT à construire.
Sortie	Le problème SAT correspondant à ϕ_n^S.	

1: $f[] \leftarrow$ tableau de taille n vide.
2: $d[][][] \leftarrow$ tableau de taille $n \times |\Sigma| \times n$ vide.
3: $x[][] \leftarrow$ tableau de taille $|prefixTree| \times n$ vide.
4: $y[] \leftarrow$ tableau de taille $|Uni|$ vide.
5: $z[] \leftarrow$ tableau de taille $|Ex|$ vide.
6: $nVar \leftarrow 0$
7: $solver(\bar{f}, \bar{d}, \bar{x}, \bar{y}, \bar{z}) \leftarrow$ solver pour problème SAT

On commence par initialiser les tableaux f, d, x, y et z qui vont contenir les variables boléennes composant le problème SAT, $nVar$ qui va permettre d'attribuer un identifiant unique à chacune de ces variables et $solver$ l'objet qui va contenir toutes les variables et contraintes boléennes de notre problème SAT.

Concrètement, le solveur SAT ne sera pas implémenté dans le cadre de ce travail. Les solveurs SAT étant des outils complexes dont il existe une multitude d'implémentations optimisées. Le solveur théorique $solver$ que nous allons décrire va cependant refléter l'usage d'un solveur SAT en pratique.

La formulation $solver(\bar{f}, \bar{d}, \bar{x}, \bar{y}, \bar{z})$ indique que ce solveur utilisera les variables de f, d, x, y et z.

Chaque variable boléenne sera identifiée par un entier unique $b > 0$. On ajoutera des contraintes φ au solveur en appelant $solver.constraint(\varphi)$. Une contrainte pour ce solveur sera une formule boléenne φ respectant la grammaire 4.1.
De plus, la fonction $solver.check()$ vérifiera s'il est possible d'assigner une valeur boléenne à chaque variable b de manière à rendre toutes les contraintes φ vraies (i.e. si le problème SAT est satisfaisable). Cette opération correspond à la ligne 6 dans $learner.conjecture$.
S'il est satisfaisable, la fonction $solver.model()$ retournera un modèle satisfaisant toutes les contraintes ajoutées au solveur.

$< variable >$	$\rightarrow$	$[0-9]+$
$< operateur\text{-}binaire >$	$\rightarrow$	$\Longrightarrow$
		$\mid \vee$
		$\mid \wedge$
$< operateur\text{-}unaire >$	$\rightarrow$	$\neg$
$< formule >$	$\rightarrow$	$< variable >$
		$\mid vrai$
		$\mid faux$
$< formule >$	$\rightarrow$	$< variable >< operateur\text{-}binaire >< formule >$
$< formule >$	$\rightarrow$	$< operateur\text{-}unaire >< formule >$
$< formule >$	$\rightarrow$	$(< formule >)$

TABLE 4.1 – Grammaire pour les formules des contraites de $solver$

La suite de l'algorithme 4 va décrire la manière dont ces formules sont construites.

```
 8: for q allant de 0 à n − 1 do
 9:     f[q] ← nVar
10:     nVar ← nVar + 1
11: end for
12: for p allant de 0 à n − 1 do
13:     for chaque a ∈ Σ do
14:         for q allant de 0 à n − 1 do
15:             d[p][a][q] ← nVar
16:             nVar ← nVar + 1
17:         end for
18:     end for
19: end for
20: for u allant de 0 à |prefixTree| − 1 do
21:     for q allant de 0 à n − 1 do
22:         x[u][q] ← nVar
23:         nVar ← nVar + 1
24:     end for
25: end for
```

On initialise les variables boléennes de type f_q, $d_{p,a,q}$ et $x_{u,q}$ en leur attribuant à chacune un identifiant entier unique.

```
26: for p allant de 0 à n − 1 do
27:     for chaque a ∈ Σ do
28:         total ← faux
29:         for q allant de 0 à n − 1 do
30:             total ← total ∨ d[p][a][q]
31:             for q2 allant de 0 à n − 1 do
32:                 solver.constraint(¬(d[p][a][q] ∧ d[p][a][q2]))        ▷ 4.1
33:             end for
34:         end for
35:         solver.constraint(total)                                      ▷ 4.2
36:     end for
37: end for
38: solver.constraint(x[prefixTree.racine.id][0])                         ▷ 4.3
39: for q allant de 1 à n − 1 do
40:     solver.constraint(¬x[prefixTree.racine.id][q])
41: end for
```

On commence par fixer les contraintes qui vont rendre l'AFD déterministe et sa fonction de transition totale. Ensuite, on va fixer l'état initial de cet AFD comme étant q_0 en contraignant le parcours de cet AFD par le mot vide (mot contenu dans la racine de l'arbre des préfixes) à arriver sur l'état 0. On s'assure ensuite que ce soit le seul parcours valide par le mot vide ϵ en contraignant les autres variables $x_{\epsilon,q\in\{1,\dots,n-1\}}$ à être fausses.

```
42: prefixes[] ← liste vide
43: Ajouter prefixTree.racine dans prefixes
44: while |prefixes| > 0 do
45:     prefix ← prefixes[0]
46:     Retirer prefix de prefixes
47:     for q allant de 0 à n − 1 do
48:         for q2 allant de 0 à n − 1 do
49:             solver.constraint(¬(x[prefix.id][q] ∧ x[prefix.id][q2]))  ▷ 4.4
50:         end for
51:     end for
52:     for chaque a ∈ Σ do
53:         if prefix à un successeur pour a then
54:             succa ← prefix.succ[a]
```

```
55:             for p allant de 0 à n − 1 do
56:                 for q allant de 0 à n − 1 do
57:                     solver.constraint((x[prefix.id][p] ∧ d[p][a][q]) ⟹
    x[succa.id][q]))                                                    ▷ 4.5
58:                 end for
59:             end for
60:             Ajouter prefix.succ[a] à prefixes
61:         end if
62:     end for
63:     if prefix.accept > 0 then
64:         for q allant de 0 à n − 1 do
65:             solver.constraint(x[prefix.id][q] ⟹ f[q])               ▷ 4.6
66:         end for
67:     else if prefix.accept < 0 then
68:         for q allant de 0 à n − 1 do
69:             solver.constraint(¬(x[prefix.id][q] ⟹ f[q]))            ▷ 4.7
70:         end for
71:     end if
72: end while
```

Ensuite, l'algorithme va parcourir tous les noeuds de l'arbre des préfixes afin de contraindre le parcours de l'AFD par les préfixes de W.
Premièrement, chaque parcours de l'arbre par un mot $u \in Pref(W)$ sera rendu unique en contraignant la vérité des variables x. Pour chaque préfixe, une seule variable $x_{prefix,q}$ pourra être rendue vraie.
Ensuite, on contraint la manière dont l'AFD sera parcouru par un mot. Si une paire de variables $x_{u,p}$ et $d_{p,a,q}$ est vraie, alors la variable $x_{ua,q}$ sera rendue vraie elle aussi. Ainsi, dans l'AFD, si on arrive sur un état q en lisant le mot u, et qu'il existe une transition $\delta(p, a) = q$, alors l'état q sera atteint en lisant le mot ua.
Pour finir, l'acceptance du mot contenu dans le noeud de l'arbre des préfixes actuel est contrainte. Si la valeur de $accept = 1$, alors on s'assure que si le parcours de l'AFD par le préfixe u de ce noeud se termine sur l'état q, alors $f_q = vrai$ ce qui fera de q un état final. La négation de cette contrainte sera appliquée si $accept = -1$.

Il reste ensuite à créer les contraintes pour les contre-exemples d'implication.

73: **if** $|S.Uni| > 0$ **then**
74: **for** u allant de 0 à $|S.Uni|$ **do**
75: $(v, \mathcal{A}) \leftarrow Uni[u]$
76: $na \leftarrow$ nombre de sommets de $\mathcal{A}$
77: $y[u][][] \leftarrow$ tableau de taille $n \times na$ vide
78: **for** q allant de 0 à $n - 1$ **do**
79: **for** qa allant de 0 à $na - 1$ **do**
80: $y[u][q][qa] \leftarrow nVar$
81: $nVar \leftarrow nVar + 1$
82: **end for**
83: **end for**
84: $aStates[] \leftarrow$ tableau de taille na vide
85: Ajouter les sommets de $\mathcal{A}$ dans $aStates$ avec $aStates[0]$ sommet initial de $\mathcal{A}$
86: $solver.constraint(y[u][0][0])$ ▷ 4.8
87: **for** p' allant de 0 à $|aStates| - 1$ **do**
88: $aState \leftarrow aStates[p']$
89: **for** chaque transition $\delta^{\mathcal{A}}(aState, a) = aState2$ de $\mathcal{A}$ **do**
90: $q' \leftarrow$ indice de $aState2$ dans $aStates$
91: **for** p allant de 0 à $n - 1$ **do**
92: **for** q allant de 0 à $n - 1$ **do**
93: $solver.constraint((y[u][p][p'] \ \wedge \ d[p][a][q]) \implies y[u][q][q'])$ ▷ 4.10
94: **end for**
95: **end for**
96: **end for**
97: **end for**
98: $antecedent \leftarrow faux$
99: $consequence \leftarrow vrai$
100: $vNode \leftarrow prefixTree.add(v, 0)$ ▷ on récupère le noeud de v
101: **for** q allant de 0 à $n - 1$ **do**
102: $antecedent \leftarrow antecedent \vee (x[vNode.id][q] \wedge f[q])$
103: **for** fa états finaux de A **do**
104: $q' \leftarrow$ indice de fa dans $aStates$
105: $consequence \leftarrow consequence \wedge (y[u][q][q'] \implies f[q])$
106: **end for**
107: **end for**
108: $solver.constraint(antecedent \implies consequence)$ ▷ 4.10
109: **end for**
110: **end if**

Pour chaque contre-exemple $(v, \mathcal{A})$ de Uni, s'il y en a, on commence par initialiser l'entrée du tableau y lui correspondant et on attribue un identifiant unique aux variables qu'il contient.
Ensuite, on récupère tous les états de l'AFD $\mathcal{A}$ et on les numérote de 0 à $|\mathcal{A}| - 1$ où $|\mathcal{A}|$ est le nombre d'états de $\mathcal{A}$, avec l'état numéroté 0 étant l'état initial de $\mathcal{A}$.
Le premier appel à $solver.constraint$ indique que les parcours en parallèle de l'AFD et de $\mathcal{A}$ commencent sur la paire d'états initiaux des deux automates.
Après cela, on détermine les paires d'états atteintes lors des parcours en parallèle des deux automates comme décrit par la formule 4.9. Si une paire d'états $(p, p') \in Q \times Q^{\mathcal{A}}$ est atteinte ($y_{p,p'} = vrai$) et qu'il existe une transition par a dans les deux automates telles que $\delta(p, a) = q$ et $\delta^{\mathcal{A}}(p', a) = q'$, alors $y_{q,q'} = vrai$.
On construit ensuite la formule 4.10 et on l'ajoute au $solver$.
La suite et fin de cet algorithme explique la manière dont sont construites les formules contraignant les contre-exemples d'implication existentielle.

111: **if** $|S.Ex| > 0$ **then**
112: **for** e allant de 0 à $|S.Ex|$ **do**
113: $(v, \mathcal{A}) \leftarrow Ex[e]$
114: $na \leftarrow$ nombre de sommets de $\mathcal{A}$
115: $l_{max} \leftarrow (n * na) - 1$
116: $z[e][][][] \leftarrow$ tableau de taille $n \times na \times l_{max}$ vide
117: **for** q allant de 0 à $n - 1$ **do**
118: **for** qa allant de 0 à $na - 1$ **do**
119: **for** l allant de 0 à $l_{max} - 1$ **do**
120: $z[u][q][qa][l] \leftarrow nVar$
121: $nVar \leftarrow nVar + 1$
122: **end for**
123: **end for**
124: **end for**
125: $aStates[] \leftarrow$ tableau de taille na vide
126: Ajouter les sommets de $\mathcal{A}$ dans $aStates$ avec $aStates[0]$ sommet initial de $\mathcal{A}$
127: $solver.constraint(z[e][0][0][0])$ ▷ 4.11
128: **for** q allant de 0 à $n - 1$ **do**
129: **for** qa allant de 0 à $na - 1$ **do**
130: **if** $(q \neq 0) \vee (qa \neq 0)$ **then**
131: $solver.constraint(\neg z[e][q][qa][0])$ ▷ 4.12
132: **end if**
133: **end for**
134: **end for**

```
135:        for p' allant de 0 à |aStates| − 1 do
136:            aState ← aStates[p']
137:            for chaque transition d(aState, a) = aState2 de A do
138:                for p allant de 0 à n − 1 do
139:                    for q allant de 0 à n − 1 do
140:                        q' ← indice de aState2 dans aStates
141:                        for l allant de 0 à l_max − 2 do
142:                            solver.constraint((z[e][p][p'][l] ∧ d[p][a][q]) ⟹
    z[e][q][q'][l + 1])                                                    ▷ 4.13
143:                        end for
144:                    end for
145:                end for
146:            end for
147:            for q allant de 0 à n − 1 do
148:                for l allant de 1 à l_max − 1 do
149:                    reverse ← faux
150:                    for p2' allant de 0 à |aStates| − 1 do
151:                        aState2 ← aStates[p2']
152:                        for chaque transition δ^A(aState2, a) = aState de A
    do
153:                            for p allant de 0 à n − 1 do
154:                                reverse ← reverse ∨ (d[p][a][q] ∧
    z[e][p][p2'][l − 1])
155:                            end for
156:                        end for
157:                    end for
158:                    solver.constraint(z[e][q][q'][l] ⟹ reverse)        ▷ 4.14
159:                end for
160:            end for
161:        end for
```

Le procédé pour contraindre les contre-exemples d'implication existentielle est similaire à celui des contre-exemples d'implication universelle auquel on ajoute une dimension l correspondant à la taille du mot par lequel les paires d'états sont atteintes dans les deux automates.
La différence entre ces deux contre-exemples est la formule $reverse$. Cette formule assure que si une contrainte de la forme $(z_{p,p',l} \wedge d_{p,a,q}) \implies z_{q,q',l+1}$ a rendu la conséquence vraie, c'est que l'antécédent est vrai lui aussi. Pour cela, si une variable $z_{q,q',l} = vrai$, alors on cherche, parmi les transitions du sommet dans l'automate du contre-exemple, si l'une d'elles atteint q' en lisant mot a depuis un

état p'. Si c'est le cas, on s'assure qu'une transition similaire existe dans l'AFD à conjecturer.

```
162:        antecedent ← faux
163:        consequence ← faux
164:        vNode ← prefixTree.add(v, 0)          ▷ on récupère le noeud de v
165:        for q allant de 0 à n − 1 do
166:            antecedent ← antecedent ∨ (x[vNode.id][q] ∧ f[q])
167:            for fa états finaux de A do
168:                q' ← indice de fa dans aStates
169:                for l allant de 0 à l_max − 1 do
170:                    consequence ← consequence ∨ (z[e][q][q'][l] ∧ f[q])
171:                end for
172:            end for
173:        end for
174:        solver.constraint(antecedent ⟹ consequence)          ▷ 4.15
175:    end for
176: end if
177: return solver
```

On termine ensuite par construire la formule 4.15 pour ce contre-exemple.
Une fois tous les contre-exemples d'implication traités, $solver$ est retourné.

La suite de l'apprentissage correspondra à un appel à $solver.check()$. Si celui-ci renvoie la valeur $vrai$, on sort de la boucle d'apprentissage et on récupère un modèle pour notre problème grâce à un appel à $solver.model()$.

Ce modèle pourra ensuite être traduit en un AFD grâce à l'algorithme 5 décrivant l'appel à $learner.translatemodel$.

Algorithm 5 learner.translatemodel

Entrées $\mathfrak{M}$: Un modèle contenant assez d'information pour construire un AFD
n : Le nombre d'états dans l'AFD à construire.
Sortie Un AFD à n états correspondant à $\mathfrak{M}$

```
1: Q ← []
2: for i allant de 0 à n − 1 do
3:     initialiser le sommet q_i
4:     ajouter q_i dans Q
5: end for
```

6: $q_0 \leftarrow Q[0]$
7: $\delta \leftarrow []$
8: **for** chaque variable $d_{p,a,q}$ dans $\mathfrak{M}$ **do**
9: **if** $\mathfrak{M}(d_{p,a,q}) = vrai$ **then**
10: ajouter $(Q[p], a, Q[q])$ dans δ
11: **end if**
12: **end for**
13: $F \leftarrow []$
14: **for** chaque variable f_q dans $\mathfrak{M}$ **do**
15: **if** $\mathfrak{M}(f_q) = vrai$ **then**
16: ajouter $Q[q]$ dans F
17: **end if**
18: **end for**
19: $\mathcal{A} \leftarrow (Q, \Sigma, q_0, \delta, F)$
20: **return** $\mathcal{A}$

Afin de construire un AFD à n états cohérent avec S selon le modèle $\mathfrak{M}$, seules les variables boléennes f et d nous intéressent, comme décrit par 4.5.2. Ainsi, l'élève commence par initialiser les n états de l'AFD et définit l'état $n = 0$ comme étant l'état initial.
Ensuite, le modèle est parcouru pour les valeurs des variables d. Si une variable $d_{p,a,q} = vrai$ dans le modèle, alors on construit une transition de l'état p vers l'état q utilisant le caractère a.
Finalement, pour chaque valeur $f_q = vrai$ du modèle, l'état f est défini comme état final de l'automate.

Une fois construit, l'AFD est retourné.

C'est maintenant à l'enseignant de poursuivre l'apprentissage. La manière dont celui-ci effectue les vérifications de la définition 4.4.1 par la méthode $teacher.check$ est décrite par l'algorithme 6

Algorithm 6 teacher.check

Entrées $\mathfrak{G}_\Sigma$: Un jeu de sûreté rationnel sur Σ
$\mathcal{C}$: AFD tentant de décrire l'ensemble gagnant W_0 pour le joueur j_0 de $\mathfrak{G}_\Sigma$.
Sortie Un contre-exemple si l'une des règles de la définition d'ensemble gagnant (4.3.3) n'est pas respectée.

1: $\mathcal{B}_I \leftarrow \mathcal{A}_I \setminus \mathcal{C}$ ▷ Vérification 1 : sommets initiaux
2: **if** $\mathcal{L}(\mathcal{B}_I) \neq \emptyset$ **then**
3: $u \leftarrow anyword(\mathcal{B}_I)$
4: **return** u
5: **end if**
6: $\mathcal{B}_F \leftarrow \mathcal{C} \setminus \mathcal{A}_F$ ▷ Vérification 2 : sommets sûrs
7: **if** $\mathcal{L}(\mathcal{B}_F) \neq \emptyset$ **then**
8: $u \leftarrow anyword(\mathcal{B}_F)$
9: **return** u
10: **end if**
11: $\mathcal{B}_{E1} \leftarrow \mathcal{R}(\mathcal{T}_E)^{-1}(\mathcal{C})$ ▷ Vérification 3 : fermeture existentielle
12: $\mathcal{B}_{E2} \leftarrow \mathcal{A}_{V_0} \setminus \mathcal{B}_{E1}$
13: $\mathcal{B}_E \leftarrow \mathcal{B}_{E2} \cap \mathcal{C}$
14: **if** $\mathcal{L}(\mathcal{B}_E) \neq \emptyset$ **then**
15: $u \leftarrow anyword(\mathcal{B}_E)$
16: $\mathcal{A} \leftarrow \mathcal{R}(\mathcal{T}_E)(\{u\})$
17: **return** $(u, \mathcal{A})$
18: **end if**
19: $\mathcal{B}_{U1} \leftarrow (\mathcal{A}_{V_0} \cup \mathcal{A}_{V_1}) \setminus \mathcal{C}$ ▷ Vérification 4 : fermeture universelle
20: $\mathcal{B}_{U2} \leftarrow \mathcal{R}(\mathcal{T}_E))^{-1}(\mathcal{B}_{U1})$
21: $\mathcal{B}_U \leftarrow \mathcal{A}_{V_1} \cap \mathcal{C} \cap \mathcal{B}_{U2}$
22: **if** $\mathcal{L}(\mathcal{B}_U) \neq \emptyset$ **then**
23: $u \leftarrow anyword(\mathcal{B}_U)$
24: $\mathcal{A} \leftarrow \mathcal{R}(\mathcal{T}_E)(\{u\})$
25: **return** $(u, \mathcal{A})$
26: **end if**

La méthode $anyword(\mathcal{A})$ retourne un mot accepté par $\mathcal{A}$, choisi au hasard.

Afin de vérifier que $\mathcal{C}$ est un ensemble gagnant pour le jeu $\mathfrak{G}_\Sigma$, l'enseignant va construire divers AFN lesquels vont permettre de vérifier individuellement les quatre règles de la définition 4.3.3.

1. Pour les sommets initiaux, l'enseignant calcule un AFN $\mathcal{B}_I$ tel que $\mathcal{L}(\mathcal{B}_I) = \mathcal{L}(\mathcal{A}_I) \setminus \mathcal{L}(\mathcal{C})$. Si $\mathcal{L}(\mathcal{B}_I) \neq \emptyset$, cela signifie qu'il y a au moins un sommet initial qui ne fait pas partie de l'ensemble gagnant. Dans ce cas, l'enseignant construit un contre-exemple positif en prenant n'importe quel mot de $\mathcal{L}(\mathcal{B}_I)$ et le retourne.

2. Pour les sommets sûrs, l'enseignant calcule l'AFN $\mathcal{B}_F$ tel que $\mathcal{L}(\mathcal{B}_F) = \mathcal{L}(\mathcal{C}) \setminus \mathcal{L}(\mathcal{A}_F)$. Si $\mathcal{L}(\mathcal{B}_F) \neq \emptyset$, alors c'est qu'au moins un sommet de W_0 n'est pas inclu dans F. Dans ce cas, l'enseignant construit un contre-exemple négatif en prenant n'importe quel mot de $\mathcal{L}(\mathcal{B}_F)$ et le retourne.

3. En ce qui concerne la fermeture existentielle, plusieurs AFN sont nécessaires.

- Premièrement, l'enseignant calcule l'AFN $\mathcal{B}_{E1}$ qui est l'image de $\mathcal{C}$ par la relation inverse $\mathcal{R}(\mathcal{T}_E)^{-1}$. Ainsi, $\mathcal{L}(\mathcal{B}_{E1})$ contient tous les sommets qui ont un successeur dans $\mathcal{L}(\mathcal{C})$.
- Ensuite, $\mathcal{B}_{E2}$ est construit de manière que $\mathcal{L}(\mathcal{B}_{E2}) = \mathcal{L}(\mathcal{A}_{V_0}) \setminus \mathcal{L}(\mathcal{B}_{E1})$. On aura donc que $\mathcal{B}_{E2}$ contient tous les sommets appartenant à j_0 qui n'ont pas de successeurs dans $\mathcal{C}$.
- Pour finir, afin d'obtenir tous les sommets de j_0 qui sont dans $\mathcal{C}$ mais qui n'ont eux-mêmes pas de successeurs dans $\mathcal{C}$, l'enseignant construit l'AFN $\mathcal{B}_E$ tel que $\mathcal{L}(\mathcal{B}_E) = \mathcal{L}(\mathcal{C}) \cap \mathcal{L}(\mathcal{B}_{E2})$.

Si $\mathcal{L}(\mathcal{B}_E) \neq \emptyset$, alors il existe un sommet de j_0 qui contredit la fermeture existentielle. L'enseignant construit un contre-exemple d'implication existentielle en prenant un mot $u \in \mathcal{L}(\mathcal{B}_E)$ et en calculant $\mathcal{A} = \mathcal{R}(\mathcal{T}_E)(\{u\})$, l'image de ce mot par la relation $\mathcal{R}(\mathcal{T}_E)$. Il retourne ainsi $(u, \mathcal{A})$ signifiant qu'un sommet u appartenant à j_0 est inclu dans l'ensemble gagnant sans qu'au moins un de ses successeurs (les mots de $\mathcal{L}(\mathcal{A})$) n'ait été accepté.

4. La fermeture universelle nécessite aussi trois étapes.

- L'enseignant commence par calculer l'automate $\mathcal{B}_{U1} = (\mathcal{A}_{V_0} \cup \mathcal{A}_{V_1}) \setminus \mathcal{C}$, l'AFN qui décrit tous les sommets de $\mathfrak{G}_\Sigma$ qui ne sont pas dans $\mathcal{C}$.
- Ensuite, en prenant l'image de $\mathcal{B}_{U1}$ par la relation inverse $\mathcal{R}(\mathcal{T}_E)^{-1}$, l'enseignant obtient $\mathcal{B}_{U2}$ qui décrit tous les sommets ayant au moins un successeur qui n'appartient pas à $\mathcal{C}$.
- Enfin, l'intersection de $\mathcal{B}_{U2}$ et de $\mathcal{C}$ permet d'obtenir tous les sommets de $\mathcal{C}$ qui ont au moins un successeur hors de $\mathcal{C}$. En ajoutant l'intersection avec $\mathcal{A}_{V_1}$, on obtient l'AFN $\mathcal{B}_U$ qui décrit tous les sommets du joueur j_1 qui ont un successeur hors de $\mathcal{C}$.

Si $\mathcal{L}(\mathcal{B}_U) \neq \emptyset$, chaque mot $u \in \mathcal{L}(\mathcal{B}_U)$ contredit la fermeture universelle. L'enseignant construit et retourne donc un contre-exemple d'implication universelle $(u, \mathcal{A})$ en prenant un de ces mots u et $\mathcal{A}$ son image par la relation $\mathcal{R}(\mathcal{T}_E)$.Ce contre-exemple signifie qu'un sommet u appartenant à j_1 est inclu dans l'en-

semble gagnant sans que tous ses successeurs y soient inclus.

Si aucun contre-exemple n'a été retourné, alors l'apprentissage s'arrête. $\mathcal{C}$ décrit W_0 et est retourné.

Cependant, s'il n'existe pas d'ensemble gagnant pour le joueur j_0. L'aglorithme d'apprentissage ne s'arrêtera jamais. Une manière de contourner cela serait d'introduire une fonction pour l'enseignant afin qu'il vérifie si l'échantillon S contient une contradiction, ce qui permettrait d'affimer la victoire dans tous les cas du joueur j_1 (i.e. $W_0 = \emptyset$). Néanmoins, cette condition ne suffirait pas à empêcher l'algorithme de continuer à l'infini car il est probable que l'échantillon S reste sans contradictions malgré qu'un ensemble gagnant ne puisse pas être appris pour le joueur j_0.

4.6.3 Complexité

Nous allons maintenant discuter la complexité des algorithmes présentés à la section précédente.
Supposons qu'un ensemble gagnant pour le joueur j_0 d'un jeu de sûreté sur un alphabet de taille a existe et soit accepté par un AFD ayant au minimum n états. Alors $Apprentissage$ (2) pour ce jeu de sûreté possède une complexité en temps de $O(a(n!)(2^{an^2}))$.

Preuve. Soit un jeu de sûreté rationnel $\mathfrak{G}_\Sigma = (A_\Sigma, \mathcal{A}_I, \mathcal{A}_F)$ sur Σ, avec $A_\Sigma = (\mathcal{A}_{V_0}, \mathcal{A}_{V_1}, \mathcal{T}_E)$. On suppose qu'il existe un ensemble gagnant pour ce jeu, dont le langage est accepté par un AFD possédant n états. Soit $S = (Pos, Neg, Ex, Uni)$ l'échantillon pour l'apprentissage. Soit $|prefixTree|$ le nombre de noeuds dans l'arbre des préfixes, $a = |\Sigma|$, $|Q_E|$ (resp. $|Q_U|$) le nombre total d'états dans les AFD des conséquences des contre-exemples d'implication universelle (resp. existentielle) de Uni (resp. de Ex). Décomposons $Apprentissage$ (2) l'algorithme de la même manière qu'à la section précédente.

- Les lignes **1-2** n'ajoutent pas de complexité à l'algorithme étant donné qu'il s'agit de déclarations de variables et sont donc en $O(1)$.
- L'algorithme entre ensuite dans sa boucle principale. La condition d'arrêt de cette boucle s'évalue en $O(1)$.
- Une itération de cette boucle commence par un appel à $learner.conjecture$. C'est cet appel qui va être responsable de la plus grande partie de la complexité de cet algorithme. Premièrement, on appelle $learner.constructSAT$. La complexité totale de cet appel est calculée comme suit :

- L'initialisation des variables se fait en $O(n+n^2a+n|prefixTree|+n|Q_U|+n^2|Q_E|^2) = O(n^2a)$ car il s'agit de parcourir des tableaux des tailles respectives n, an^2, $n|prefixTree|$, $n|Q_U|$ et $n^2|Q_E|^2$ et d'y effectuer des assignations en $O(1)$
- La contrainte des variables d (lignes **26-37**) se fait en $O(an^2)$.
- La contrainte des variables x initiales (lignes **38-41**) se fait en $O(n)$
- La contrainte des variables x pour tous les préfixes dans $prefixTree$ (lignes **42-72**) se fait en temps $O(|prefixTree|*(n^2*an^2*n*n)) = O(an^2)$.
- Les contraintes sur les contre-exemples d'implication universelle (lignes **84-107**) se contruisent en temps $O(|Q_U|*(|Q_U|^2*a*n^2+n|Q_U|)) = O(an^2)$. En effet, étant donné que la boucle ligne **73-110** itère sur chaque sommet du contre-exemple actuel, on peut considérer qu'elle itère sur tous les sommets de tous les contre-exemples dans Uni, par la mise en évidence.
  Ensuite, la première boucle interne étant l'initialisation des variables y, elle a déjà été prise en compte dans notre calcul.
  La boucle suivante (lignes **87-97**) itère d'abord sur les n, puis sur les transitions de l'état actuel dans $|Q_U|$, dont on peut borner le nombre total à $|Q_U|*a$ car il y a au plus a transitions qui démarrent de chaque sommets. Ensuite, les deux boucles les plus imbriquées itèrent à nouveaux sur les n. Pour finir, la dernière étape (lignes **101-107**) utilise deux boucles imbriquées lesquelles itèrent respectivement sur toutes les valeurs de n et de $|Q_U|$ (car, dans le pire des cas, tous les états de tous les automates de tous les contre-exemples sont des états finaux).
- Les contraintes pour les contre-exemples d'implication existentielle se construisent, par raisonnement similaire au point précédent avec un facteur $l_{max} = n * |Q_E|$ additionel, en temps $O(|Q_E| * (n|Q_E| + n^3a|Q_E|^3 + n^3a|Q_E|^4) + n^2|Q_E|^2) = O(an^3)$

Ainsi, un appel à la fonction $learner.conjecture$ aura une complexité de $O(an^3)$. Le nombre $nVar$ de variables boléennes créées par cet algorithme sera de $nVar = n+n^2a+n|prefixTree|+n|Q_U|+n^2|Q_E|^2 \approx an^2$

- La méthode $learner.solveSAT$, correspondant à l'utilisation d'un solveur SAT comme décrit dans la section précédente, va tenter d'attribuer une valeur, parmi deux possibles, à chaque variable créée dans le problème SAT. Ainsi, cet appel aura pour complexité $O(2^{nVar}) = O(2^{an^2})$.
- De retour à la boucle principale, la méthode $teacher.check$ aura pour complexité la somme des opérations successives réalisées sur les automates. Ainsi, étant donné que la complexité d'une somme est équivalente

à la complexité de son terme dominant, nous n'avons qu'à considérer la complexité de l'opération la plus lourde réalisée par l'enseignant dans cet algorithme. Cette opération est la dernière étape de la vérification de la fermeture existentielle. Calculer l'intersection de deux AFN $\mathcal{A}$ et $\mathcal{B}$ avec $|\mathcal{A}| = n_a$ et $|\mathcal{B}| = n_b$ se fait en temps $O(n_a n_b)$. De ce fait, même dans le pire des cas, les vérifications de l'enseignant n'influenceront pas la complexité totale de la boucle de notre algorithme, car on a déjà calculé une complexité en $O(2^{an^2})$ qui bornerait la complexité quadratique de cette étape.

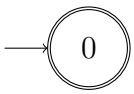
— Pour finir, la complexité de la méthode $translatemodel$ est linéaire pour le nombre de variable créées dans le modèle, et est donc en $O(nVar) = O(an^2)$.

La dernière étape pour ce calcul de complexité est de se demander combien de fois au maximum la boucle principale va s'exécuter. Pour un nombre d'états n et un alphabet de taille a, il y a au plus $(a+1)n!$ configurations d'AFD. Ainsi, dans le pire des cas, la boucle principale s'exécutera $\approx an!$ fois.
La complexité totale de l'algorithme d'apprentissage s'élève donc à $O(1 + (an!) * (an^3 + 2^{an^2}) + an^2) = O(a(n!)(2^{an^2}))$. □

4.6.4 Exemple

Afin d'illustrer le fonctionnement de cet algorithme, nous allons reprendre l'exemple de jeu de sûreté rationnel sur l'alphabet $\Sigma = \{e, s, |\}$ introduit à la section 4.2 et décrire l'échange entre un enseignant et un élève pour ce jeu dans le but d'apprendre un ensemble gagnant W_0 pour le joueur j_0.

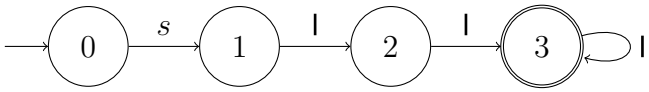
Après avoir initialisé un échantillon S et un arbre des préfixes $prefixTree$, vides tous les deux, l'élève cherche à conjecturer un premier AFD à $n = 1$ états. Etant donné qu'il n'y a pas de contre-exemple dans S, l'élève retourne l'AFD suivant :



Avec $\mathcal{L}(\mathcal{C}) = \emptyset$. L'enseignant, confronté à cet AFD, va retourner un contre-exemple positif car cet AFD ne vérifie pas la première condition de la définition 4.3.3 d'ensemble gagnant. En effet, si l'on soustrait le langage de $\mathcal{C}$ au langage de $\mathcal{A}_I$, on conserve l'intégralité du langage de ce dernier. L'enseignant choisit alors un mot dans $\mathcal{A}_I$ et le retourne. Disons que le contre-exemple positif retourné est $s||$. Celui-ci sera ajouté à l'échantillon, dans Pos ainsi qu'à l'arbre des préfixes,

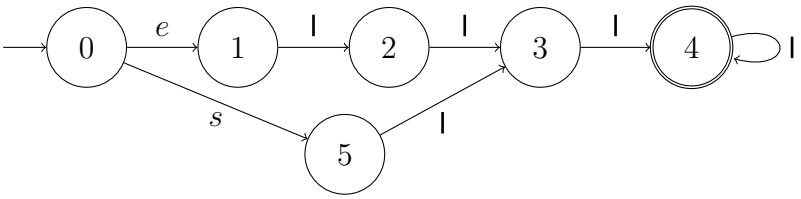
avec une valeur $accept = 1$. Cet arbre contient maintenant 4 noeuds.

Ensuite, lors de l'itération suivante, supposons que l'élève conjecture l'AFD suivant, cohérent avec S.



Cet AFD va respecter toutes les conditions de la définition 4.3.3 sauf la fermeture existentielle. L'enseignant renvoie donc un contre-exemple d'implication existentielle $(s||, \mathcal{A})$ avec $\mathcal{L}(\mathcal{A}) = \{e||, e|||\}$. En effet, l'AFD conjecturé par l'éleve ne contient pas de sommet de l'enseignant. Celui-ci lui affirme donc, au travers du contre-exemple, que si le sommet $s||$ veut être accepté dans l'ensemble gagnant, alors il faut qu'au moins un des sommets de l'environnement parmi $e||$ et $e|||$ soit accepté. (D'ailleurs, $s||$ faisant partie de Pos, il sera accepté par tous les futurs AFD conjecturés par l'élève, et fera donc partie de l'ensemble gagnant s'il existe).

Finalement, l'AFD suivant, cohérent avec les deux contre-exemples précédents, sera conjecturé par l'élève :



Cet AFD respecte les 4 règles de la définition d'ensemble gagnant. L'enseignant ne retourne pas de contre-exemple et l'apprentissage s'arrête.

Conclusion

Dans ce mémoire, nous avons étudié la manière de résoudre les jeux de sûreté joués sur des arènes de taille finie et infinie.

Dans le premier cas, nous avons pu implémenter un algorithme simple qui nous a permis de calculer une stratégie gagnante pour le système. En effet, en calculant un attracteur pour les sommets hors de l'ensemble des sommets sûrs, le système peut gagner la partie en ne déplaçant jamais le pion vers un des sommets de cet attracteur.

Dans le cas infini, nous n'avons été capables que de décrire un algorithme partiel. Munis d'une définition plus robuste de l'ensemble gagnant, il s'agit de transformer le jeu de sûreté en un jeu de sûreté rationnel, en utilisant les définitions de langages réguliers et de relations régulières. Ensuite, une étape d'apprentissage entre un élève et un enseignant pour ce jeu de sûreté rationnel permettra d'apprendre un ensemble gagnant pour le système. Pour cela, l'élève construit une série de problème SAT et les résout, afin d'en déduire un automate acceptant le langage qui décrit les sommets de l'ensemble gagnant du système pour le jeu.
Cet algorithme n'étant que partiel, il ne s'arrête pas si un tel ensemble n'existe pas. Ce potentiel ensemble gagnant donne immédiatement une stratégie pour le jeu infini. Il suffit au système de toujours déplacer le pion vers un des sommets de cet ensemble. S'il n'existe pas, le système ne peut pas garantir le respect de la spécification.

Dans un futur travail, cet algorithme pourrait être amélioré en ajoutant des moyens à l'enseignant de détecter qu'un ensemble gagnant n'existe pas et ainsi mettre fin à l'apprentissage. Sa complexité aussi pourrait être améliorée, par exemple, en augmentant le nombre d'état de l'automate à conjecturer de manière plus intelligente après chaque itération où un problème SAT construit par l'élève n'était pas satisfaisable.

Bibliographie

[1] D. Neider. Applications of automata learning in verification and synthesis. Technical report, 2014.

[2] D. Neider and U. Topcu. An automaton learning approach to solving safety games over infinite graphs. In M. Chechik and J.-F. Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 204–221, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[3] W. Thomas. Church's problem and a tour through automata theory. Technical report, RWTH Aachen, Lehrstuhl Informatik 7, 52056 Aachen, Germany, 2008.