École Centrale de Nantes

MPAR - PROJET
Rapport

Projet de MPAR

Élèves :
Clément Auclin
Félix Doublet

Enseignant :
Benoît Delahaye

2 avril 2023

Table des matières

1 Présentation du projet **2**

2 Représentation et simulation de chaînes de Markov **2**
2.1 Structure de données . . . 2
2.2 Affichage . . . 2
2.3 Lancement du programme . . . 3
2.4 Simulation . . . 5
2.4.1 Sans adversaire . . . 5
2.4.2 Random . . . 5
2.4.3 Avec adversaire . . . 5
2.4.4 Affichage de la simulation . . . 6
2.5 Gestion des récompenses . . . 6

3 Model Checking **6**
3.1 Model Checking probabiliste . . . 6
3.1.1 Until pour une chaîne de Markov . . . 7
3.1.2 Until pour un MDP . . . 7
3.2 Model Checking Statistique - SMC . . . 8
3.2.1 SMC quantitatif . . . 8
3.2.2 SMC qualitatif . . . 9

4 Calcul de l'adversaire optimal **10**
4.1 Algorithme d'itération de valeurs . . . 10
4.2 Algorithme de Q-learning . . . 11

5 Conclusion **12**

1 Présentation du projet

Le projet se décompose en deux parties afin de construire un model-checker probabiliste et statistique pour les modèles probabilistes discrets :

— Dans un premier temps, être capable d'interpréter, d'afficher et de simuler des chaînes de Markov et des MDP (Markov Decision Process).
— Dans un second temps, implémenter des algorithmes de model checking et SMC (Statistical Model-Checking) de chaînes de Markov.
— On utilisera finalement l'apprentissage par renforcement pour trouver l'adversaire optimal.

2 Représentation et simulation de chaînes de Markov

2.1 Structure de données

Afin de représenter au mieux la structure de données des MDP et des DTMC, nous avons choisi de représenter dans notre programme les différentes informations sous la forme de dictionnaires. Plus spécifiquement, après lecture d'une MDP ou d'une DTMC en .mdp, nous récupérons les informations structurées sous forme d'un dictionnaire dont les clés sont :

— 'States' comportant les états du fichier.
— 'Actions' correspondant aux actions de la chaîne.
— 'Transitions_with_action' pour les transitions avec actions. Afin d'expliciter ces données, chaque transition est elle-même un dictionnaire permettant de savoir l'état de départ, l'état d'arrivée, les poids et les actions de la transition en question.
— 'Transitions_without_action' pour les transitions sans actions. Chaque transition sans action est elle-même un dictionnaire.

Cette structure de données à plusieurs avantages. Premièrement, elle permet un accès facilité à l'ensemble des informations qui sont dans la même structure (que l'on peut par exemple nommée "chaine"). En effet, chaque type d'information pourra être recherché par son nom. Il est facile par exemple d'accéder pour une transition à ces actions en utilisant la clé du même nom. Le second avantage est la lisibilité même du code : en accédant à l'ensemble des informations par les clés, il est simple de comprendre et d'écrire les différentes fonctions, sans avoir à passer plus de temps que nécessaire sur la récupération des données dont nous avons besoin.

L'ensemble des fonctions nécessaires au cours de ce TP sont écrites dans une même classe nommée markov().

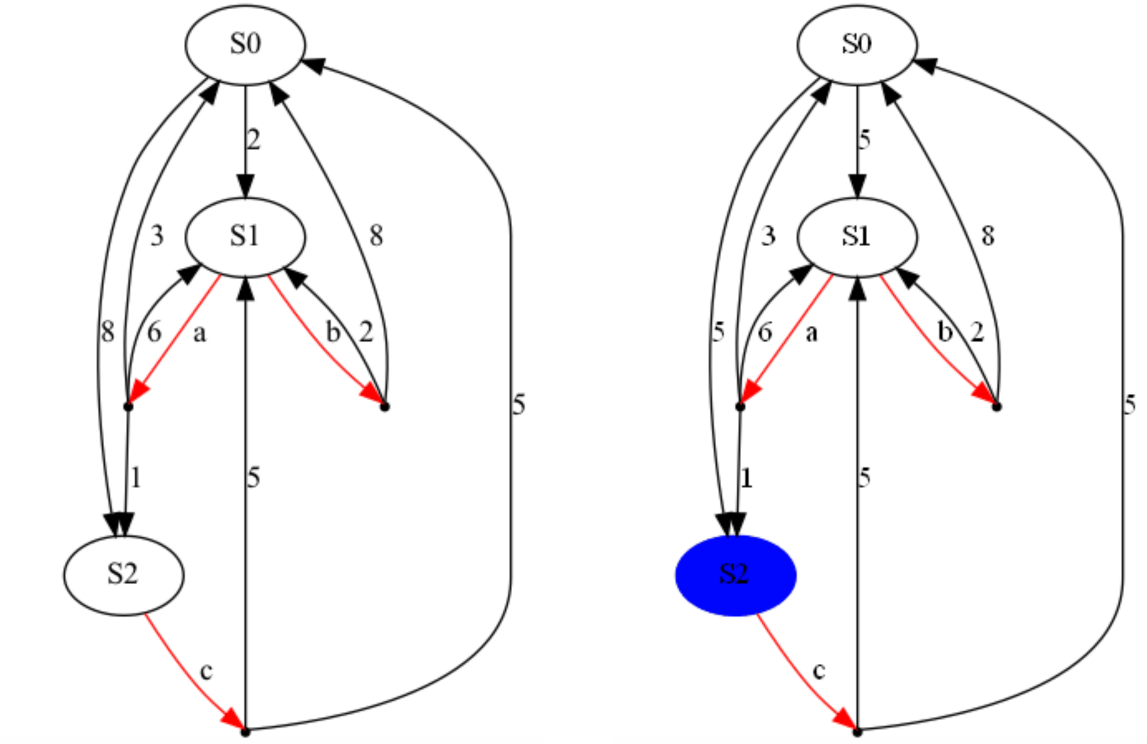
2.2 Affichage

L'affichage des chaines de Markov ou des MDP s'effectue à l'aide de la fonction *afficher* :

```
def afficher(self, etat = "default", file_name = "graph")
```

On crée un objet Digraph grâce à la bibliothèque Graphviz puis on lui associe les States récupérés. Ensuite, on crée les arêtes pour les états qui ne possèdent pas d'action, avant de faire de même pour les états qui possèdent des actions.

La variable "etat" joue sur l'affichage graphique :
— Si etat = "default", le graphe est affiché sans distinction entre les états
— Si etat = S, avec S un sommet, alors le sommet S est coloré en bleu. Cette fonctionnalité sera utile pour la simulation.



(a) Une MDP avec etat="default" (b) Le même graphe avec etat="S2"

FIGURE 1 – Affichages obtenus pour une MDP

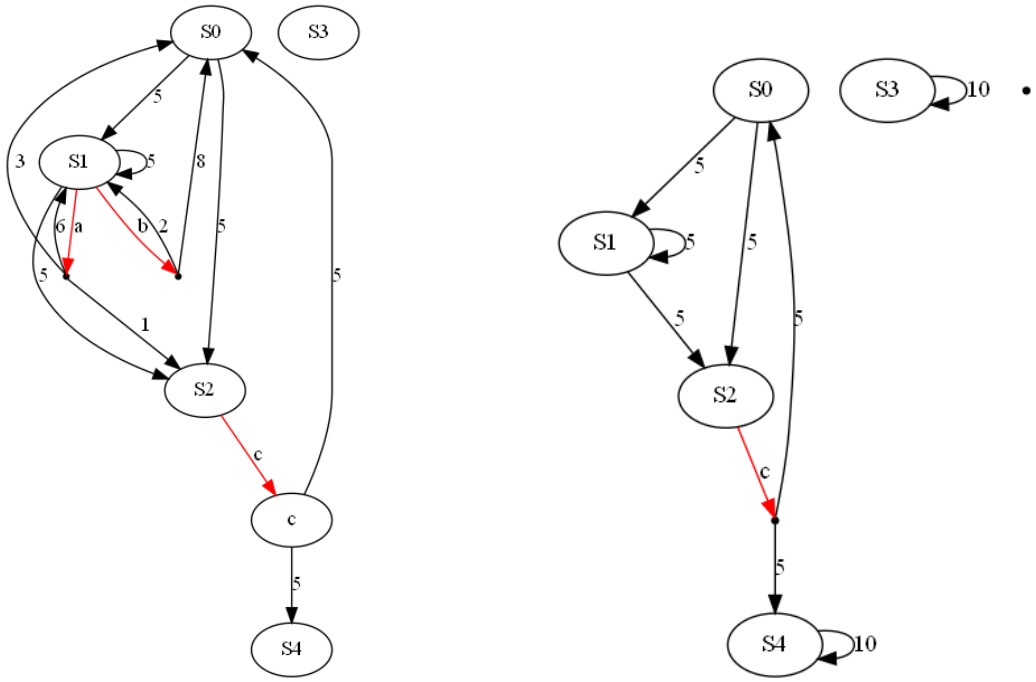
2.3 Lancement du programme

Au lancement du programme, et après la récupération des données et la création des structures correspondantes, il est demandé à l'utilisateur s'il souhaite que le programme vienne automatiquement corriger les erreurs détectées.

En effet, après la création des structures et avant toute autre action, le programme vérifie que la DTMC ou la MDP est correctement définie, et peut venir corriger la plupart des erreurs :
— On vérifie que les états déclarés sont utilisés, et on vient rajouter une transition de l'état vers lui-même si ce n'est pas le cas
— On vérifie qu'un état utilisé dans une transition est déclaré, et si ce n'est pas le cas, on déclare cet état.
— On vérifie qu'un état n'est pas déclaré plusieurs fois
— On vérifie que chaque état possède bien une transition de sortie. Si ce n'est pas le cas, on rajoute une transition de l'état vers lui-même.
— On vérifie que les actions ne sont pas déclarés plusieurs fois
— On vérifie que les actions déclarées sont utilisées et si ce n'est pas le cas, on supprime tout simplement les actions inutilisées.

— On vérifie que les actions utilisées sont déclarées et on vient rajouter les actions manquantes le cas échéant.
— On vérifie qu'un état ne possède pas à la fois des transitions sans et avec actions

Afin de me visualiser les changements que le programme peut effectuer, on représente dans la figure ci-dessous un MDP mal défini et non corrigé, et le même MDP mais corrigé par le programme :



(a) MDP avant correction (b) MDP après correction

FIGURE 2 – Correction automatique des chaînes

Voici de plus les erreurs affichées pour la chaine ci-dessus :

```
Warning : l'état S1 comporte des transitions avec et sans action

Warning : l'état S1 comporte des transitions avec et sans action
Warning : l'état S4 est utilisé dans une transition mais n'est pas déclaré
Warning : l'état S3 est déclaré mais n'est pas utilisé
Warning : un état est déclaré plusieurs fois
Warning : les états suivants n'ont pas de transition de sortie :  ['S3']
Rajout d'une transition des états ['S3'] vers eux mêmes
Warning : une action est déclarée plusieurs fois
Warning : l'action c est utilisée mais n'est pas déclarée

Écrire ok pour contiuer
-------------------------
```

FIGURE 3 – Erreurs affichées

Afin de s'assurer que l'utilisateur a bien pris connaissance des warnings, il doit rentrer "ok" pour que le programme passa à la suite.

Ensuite, sont affichés par l'utilisateur l'ensemble des choix qu'il peut faire et qui sont les suivants :

— lancer un parcours
— Effectuer le model checking d'un eventually ou d'un next
— Effectuer du SMC quantitatif ou qualitatif
— Utiliser un algorithme d'itération de valeurs
— Utiliser un algorithme de Q-learning

2.4 Simulation

Maintenant que notre structure de données est bien définie, que l'on peut afficher une chaîne de Markov ou un MDP, il ne reste plus qu'à tout assembler pour réaliser une simulation effective, i.e un parcours d'une chaîne.

2.4.1 Sans adversaire

Nous avons d'abord configuré une méthode dans la classe permettant le parcours d'une DTMC où aucun adversaire n'est requis. L'utilisateur peut ensuite rentrer le nombre d'étapes qu'il veut effectuer. Afin d'éviter toute erreur, un *Warning* est affiché lorsqu'une action est détectée alors que ce mode est enclenché, l'utilisateur peut alors soit terminer le programme, soit changer de mode. Le calcul lui-même du parcours est assez simple : on vient à l'état actif (celui sur lequel on se trouve en ce moment) regarder l'ensemble des états qui sont atteignables ainsi que leur probabilité, après une normalisation pour que les probabilités soient entre 0 et 1, on utilise la fonction *random* du module du même nom pour choisir l'état dans lequel on va se rendre.

2.4.2 Random

Ensuite, si l'utilisateur ne souhaite pas préciser un adversaire positionnel, il peut choisir le mode random qui choisit à chaque état possédant des actions l'une d'entre elles de manière uniforme.

Dans ce cas, si l'état actuel ne possède pas d'action, on choisit en respectant les transitions un nouvel état (de manière similaire au mode sans adversaire). Dans le cas contraire, on liste les actions disponibles pour cet état, on en choisit une de manière uniforme et on se ramène alors dans le cas précédent.

Cette fonction random permet de rapidement visualiser les états atteignables sans avoir à s'embêter à préciser les adversaires.

2.4.3 Avec adversaire

Le parcours avec adversaire s'effectue sur le même principe que celui sans. L'utilisateur peut choisir deux types de parcours : soit en utilisant un adversaire positionnel, soit en utilisant un adversaire qui ne l'est pas. Si c'est le choix d'un adversaire positionnel qui est fait, alors l'utilisateur va être invité avant le parcours à spécifier les actions que l'adversaire va prendre pour chaque état de la manière suivante :

```
Choix d'un adversaire positionnel
-------------------------------------------------
| Etat actif : S1                               |
| choix possibles pour l'état S1:               |
| ['b', 'a']                                    |
| choix de l'action  :                          |
| Vous avez choisi l'action b                   |
-------------------------------------------------
```

FIGURE 4 – Choix des actions pour l'état S1

Si cependant l'utilisateur choisit un adversaire non positionnel, il aura à faire ce choix à chaque action tout au long du parcours. Il est ainsi recommandé d'utiliser le parcours *random* comme vu précédemment afin de ne pas avoir à remplir à chaque étape du parcours les actions.

2.4.4 Affichage de la simulation

La fonction "afficher" décrite précédemment permet d'obtenir assez simplement la simulation. À chaque changement d'état, on applique la fonction avec le paramètre etat qui correspond à l'état courant. Une fois le parcours effectué, on concatène les images afin de générer un gif. On peut alors supprimer automatiquement toutes ces images pour ne garder que le gif qui retrace correctement la succession des états.

Le parcours est alors disponible dans le fichier "visu_parcours.gif".

2.5 Gestion des récompenses

Il a été implémenté un système de récompense dans le programme. Plus précisement, la grammaire a été modifiée pour pouvoir accepter directement des chaines avec les récompenses pour chaque état. On enregistre alors pour chacun des états leur récompense à l'aide de notre structure de données. Si aucune récompense n'est donnée, alors on remplit la structure de données pour les récompenses par des 0.

L'utilisation la plus simple de celles-ci sont lors des parcours. À chaque fin de parcours, est affiché la récompense totale obtenue. Cependant, les rewards ne servent pas qu'à ceci. En effet, ils sont nécessaires dans les algorithmes d'itération de valeurs et de Q-learning que nous avons implémentés et dont nous parlerons dans autre partie.

3 Model Checking

Deux types de model checking peuvent-être distingués : le model checking statistique (SMC) et le modèle checking probabiliste.

3.1 Model Checking probabiliste

On essaie ici de calculer la probabilité d'atteindre un (ou plusieurs) état(s).

3.1.1 Until pour une chaîne de Markov

On commence par répondre à ce problème pour des chaînes de Markov. Cela est fait dans la focntion suivante qui prend pour seul paramètre la liste des états dont on veut vérifier la probabilité d'accessibilité.

```
def eventually_check_DTMC(self, S1)
```

On utilise alors un algorithme de recherche en profondeur afin d'identifier les ensembles S_0(Sommets qui ne peuvent pas mener à S_1) et $S_?$ (Sommets qui peuvent mener à S_1).

On écrit alors le système linéaire à l'aide des matrices A et b comme explicité dans le cours puis on résout l'équation $(Id - A)X = b$

Le vecteur X correspond alors à la probabilité d'atteindre S_1 à partir de chacun des sommets de $S_?$. On choisit alors la composante de X qui correspond à l'état initial pour renvoyer sa valeur.

```
Les états S_0 sont : ['Lost']
Les états S_? sont : ['start', 'S410', 'S59', 'S68']
Les états S_1 sont : ['Won']
La probabilité de respecter la propriété à partir de l'état initial start est : 0.4929292929292929
 vecteur y total :
[[0.49292929]
 [0.33333333]
 [0.4       ]
 [0.45454545]]
```

FIGURE 5 – Résultats obtenus pour le jeu de craps (avec Won)

3.1.2 Until pour un MDP

On va de nouveau construire les matrices A et b de manière similaire à ce qu'on a fait avec les chaînes de Markov, mais de manière adaptée aux MDP en prenant en compte les différentes actions.

Uns fois nos matrices correctement remplies, on utilise la fonction linprog de scipy pour trouver la solution minimale de l'inéquation $Ax \geq b$.

Voici les résultats obtenus sur l'exemple du casino légèrement modifié avec l'objectif S_4 : S_1 boucle sur lui-même afin d'avoir un, probabilité nulle d'atteindre un autre état.

```
States S0:0, S1:5, S2:100, S3:500, S4:3;
Actions a,b;
S0[a] -> 5:S1 + 5:S2;
S0[b] -> 1:S3 + 9:S4;
S1 -> 10:S1;      You, 2 weeks ago • fonc
S3 -> 10:S0;
S2 -> 10:S0;
S4 -> 10:S0;
```

FIGURE 6 – Casino modifié pour tester l'efficience du programme

```
Quel sont les états cibles ? Veuillez les rentrer un par un, et taper 'fin' quand vous avez fini
[[ 1.  -0.5 -0.5  0. ]
 [ 1.   0.   0.  -0.1]
 [-1.   0.   0.   0. ]
 [ 1.   0.   0.   0. ]
 [ 0.   0.   0.   0. ]
 [ 0.  -1.   0.   0. ]
 [ 0.   1.   0.   0. ]
 [-1.   0.   1.   0. ]
 [ 0.   0.  -1.   0. ]
 [ 0.   0.   1.   0. ]
 [-1.   0.   0.   1. ]
 [ 0.   0.   0.  -1. ]
 [ 0.   0.   0.   1. ]]
[[ 0. ]
 [ 0.9]
 [-1. ]
 [ 0. ]
 [ 0. ]
 [-1. ]
 [ 0. ]
 [ 0. ]
 [-1. ]
 [ 0. ]
 [ 0. ]
 [-1. ]
 [ 0. ]]
Bravo ! L'algorithme converge ! La probabilité d'atteindre l'ensemble à partir des autre sommets est :

Etat S0 : 1.0
Etat S1 : 0.0
Etat S2 : 1.0
Etat S3 : 1.0
```

FIGURE 7 – Résultats obtenus avec la cible S4

On aurait facilement pu adapter cet algorithme pour donner la probabilité d'accessibilité sous la contrainte d'un certain nombre de coups en redéfinissant correctement A et b à chaque itération. Cela n'a pas été fait faute de temps.

3.2 Model Checking Statistique - SMC

Dans un premier temps, une fonction annexe a été créée qui renvoie l'état final obtenu après un nombre "length" d'itérations en partant de "etat".

```
def parcours_SMC(self, length, etat)
```

Cette fonction sera réutilisée dans les deux fonctions suivantes.

3.2.1 SMC quantitatif

Nous avons implémenté l'algorithme vu en cours dans la fonction :

```
def smc_quantitatif(self)
```

La fonction demande à l'utilisateur la précision, l'erreur et la longueur jusqu'à laquelle on ira pour vérifier la propriété.

On calcule alors le nombre d'itérations conformément à la formule, puis pour chaque itération, on regarde si l'état a été atteint avant de renvoyer le ratio entre le nombre de fois où l'état a été atteint et le nombre d'itérations.

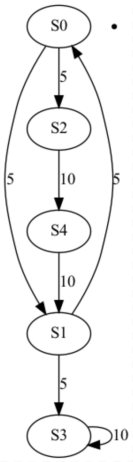


FIGURE 8 – Chaine utilisée pour illustrer les résultats

```
--------------------------------------------------
Quel sont les états cibles ? Veuillez les rentrer un par un, et taper 'fin' quand vous avez fini
Etats cibles :  ['S4']
Précision souhaitée :
Précision choisie :  0.05
Erreur souhaitée :
Erreur choisie :  0.05
Longueur des chaines voulues :
Longueur des chaines :  50
--------------------------------------------------
Calcul en cours..

Les probabilité d'atteindre chacun des états cibles ['S4'] sont : [0.71274]
```

FIGURE 9 – Résultats pour un SMC quantitatif

3.2.2 SMC qualitatif

On cherche ici à vérifier la satisfaction de la minoration de la probabilité d'atteindre un ensemble d'états plutôt qu'à chercher à la calculer.

```
def smc_quanlitatif(self)
```

Ici, on demande à l'utilisateur les valeurs alpha, bêta, thêta, epsilon et la longueur d'exploration avant d'itérer conformément aux formules du cours jusqu'à ce qu'on puisse confirmer ou infirmer si la probabilité cherchée et supérieure à thêta.

À chaque itération, on réalise une nouvelle simulation, on actualise les coefficients et s'ils permettent de conclure, l'algorithme s'arrête, sinon on continue.

L'algorithme renvoie alors la réponse à la question initiale ainsi que le nombre d'itérations nécessaires avant d'arriver à celui-ci.

```
------------------------------------------------
Quel est l'état cible ?
Etat cible :  S4
Valeur de alpha :
alpha =  0.05
Valeur de beta :
beta =  0.05
Valeur de théta :
théta =  0.65
Valeur de epsilon (zone d'indifférence) :
epsilon =  0.05
Longueur des chaines voulues :
Longueur des chaines =  50
------------------------------------------------
Calcul en cours..

La probabilité d'atteindre l'état S4 est bien supérieure ou égal à 0.65
Le nombre de simulations nécessaires pour démontrer ce résultat est de 203
```

FIGURE 10 – Résultats obtenus pour un SMC qualitatif avec la même chaîne que précedemment

4 Calcul de l'adversaire optimal

Deux algorithmes ont été implémentés et permettent de trouver les meilleures actions à effectuer pour chaque état.

4.1 Algorithme d'itération de valeurs

L'algorithme d'itération de valeurs a été le premier à être implémenté et reprend point par point l'algorithme vu en cours :

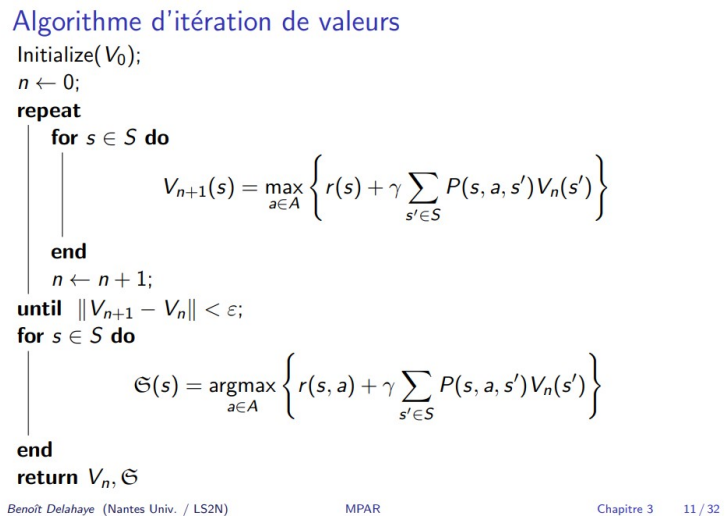


FIGURE 11 – Algorithme d'itération de valeurs

Cet algorithme a été implémenté sous la forme de deux fonctions :

```
    def algos_iterations(self)
    def calcul_max(self,etat_actif,V0,gamma)
```

La première reprend le calcul global de l'algorithme, mais c'est la deuxième fonction qui effectue à chaque itération, les maximums sur les actions pour chaque état pour plus de lisibilité. La seconde fonction est appelée par la première pour chaque état à chaque itération.

L'utilisateur est invité à fournir les valeurs voulues, c'est-à-dire la précision epsilon, la valeur de gamma pour les calculs de V, ainsi qu'une borne supérieure d'itérations dans le cas où l'algorithme ne convergerait pas.

À la fin de l'algorithme, on vient afficher différentes informations :

— La valeur finale de Vn
— Le nombre d'itérations nécessaires à la convergence
— L'adversaire optimal pour chaque état. Si un état ne possède pas d'actions, dans ce cas-là l'adversaire proposé est "None"

Voici un résumé des entrées et sorties de cet algorithme :

```
--------------------------------------------------
Choix de gamma :
Gamma choisi : 0.01
Choix de epsilon :
Epsilon choisi : 0.01
Nombre d'itérations avant arrêt forcé (au cas où il n'y pas de convergence) :
Nombre d'itérations max : 1000
--------------------------------------------------
Calcul de Vn par itérations...

--------------------------------------------------
Valeur de Vn :
[[4.045505]
 [6.051491]
 [3.05048 ]]
Nombre d'itérations nécessaires pour la couvergence (s'il y a convergence):  2
--------------------------------------------------

--------------------------------------------------
Adversaire optimal choisi (si None, c'est qu'il n'y a pas d'actions de disponibles) :
[['S0' None]
 ['S1' 'a']
 ['S2' 'c']]
--------------------------------------------------
```

FIGURE 12 – Entrées sorties de l'algorithme d'itération de valeurs

4.2 Algorithme de Q-learning

Parmis les algorithmes d'aprentissage par renforcement présentés, nous avons choisi d'implémenter l'algorithme de Q-learning.

Afin d'implémenter le plus simplement possible cet algorithme, nous avons utilisé deux fonctions distinctes :

```
def parcours_q_learning(self, etat, action_param)
def q_learning(self)
```

La première fonction permet le parcours avec les paramètres donnés par l'algorithme de Q-learning. En effet, contrairement à de nombreux algorithmes vu précédement, on n'effectue pas plusieurs simulations complètes des MDP mais on avance d'un pas à chaque itération. La fonction *parcours_q_learning* permet alors de savoir de quel état on part pour une itération donnée, et quelle action a été choisie. Elle renvoie la récompense de l'état qu'on a quitté ainsi que l'état d'arrivée.

La seconde fonction correspond à l'algorithme de Q learning lui-même et appelle donc la fonction précédente. Cette fonction demande à l'utilisateur les valeurs de différents paramètres : gamma, le nombre d'itérations total ainsi que s'il souhaite ou non à la fin

du programme que la matrice Q soit affichée en plus de l'adversaire trouvé. A chaque itération de l'algorithme de Q learning, on vient recalculer la valeur de la matrice Q pour le couple état-action on l'on se trouve. Pour cela on a besoin de plusieurs informations quant au déplacement qui suit (le reward de l'état dont on sort et l'état dans lequel on arrive), qu'on obtient grâce à la fonction *parcours_q_learning*.

De plus, nous avons implémenté comment gérer le dilemme exploration/exploitation : A chaque itération, il y a une probabilité de 0.9 que l'action choisie pour le déplacement soit celle avec la plus grande valeur dans Q pour l'état en question, et il y a donc une probabilité de 0.1 qu'une action aléatoire soit choisie. De plus, afin d'éviter tout blocage dans un état en particulier, une vérification est effectuée à chaque itération : si l'on se trouve dans un même état deux fois d'affilés et que de nouveau le déplacement nous ramène dans cet état, alors on repart du début de la chaîne.

En sortie, le programme affiche une matrice qui pour chaque état donne l'action à privilégier. Si aucune action n'est disponible pour un état, l'algorithme affiche "None".

Voici un exemple d'affichage de sortie :

```
--------------------------------------------------
Choix de gamma :
Gamma choisi : 0.01
Choix de epsilon :
Epsilon choisi : 0.01
Nombre d'itérations avant arrêt forcé (au cas où il n'y pas de convergence) :
Nombre d'itérations max : 1000
--------------------------------------------------
Calcul de Vn par itérations...

--------------------------------------------------
Valeur de Vn :
[[4.045505]
 [6.051491]
 [3.05048 ]]
Nombre d'itérations nécessaires pour la couvergence (s'il y a convergence):  2
--------------------------------------------------

--------------------------------------------------
Adversaire optimal choisi (si None, c'est qu'il n'y a pas d'actions de disponibles) :
[['S0' None]
 ['S1' 'a']
 ['S2' 'c']]
--------------------------------------------------
```

FIGURE 13 – Entrées sorties de l'algorithme d'itération de valeurs

5 Conclusion

Au cours de ce projet, nous avons réalisé un tour d'horizon des chaînes de Markov et des MDP en réalisant un programme capable de les afficher, simuler et faire du modèle-checking à partir d'une grammaire définie.

Avec plus de temps, nous aurions pu implémenter des algorithmes pour réaliser le model-cheking sur les MDP ou encore tester d'autres algorithmes d'apprentissage par renforcement.

Dans l'ensemble, le projet est tout de même une réussite puisque nous avons abouti à la création d'un outil fonctionnel et facile d'utilisation.