
Compte rendu du projet

Construction d'un model-checker probabiliste et statistique pour les
modèles probabilistes discrets

Arthur Chereau - Atys Panier

MPAR

01/04/2023

Table des matières

1	Introduction	2
2	Mise en œuvre	2
2.1	Extraction des informations sur les processus à partir des fichiers de grammaire . . .	2
2.2	Vérification	3
2.3	Implémentation de la fonction pour se déplacer de manière aléatoire dans un graphe .	3
2.4	Visualisation du parcours de graphe avec Graphviz	3
2.5	PCTL pour vérifier P(Es) sur une MC	3
2.5.1	Principe de la fonction "identifier_ensembles"	3
2.5.2	Model checking avec un parcours en profondeur	4
2.5.3	Méthode du gradient conjugué	4
2.6	Application de méthodes SMC quantitatives et qualitatives	4
2.6.1	SMC quantitative pour une MC	4
2.6.2	SMC qualitative pour une MC	4
2.7	Application des itérations de valeurs pour optimiser l'apprentissage par renforcement	5
2.8	Application de l'algorithme Q-learning	5
2.9	Résumé du travail effectué	5
3	Résultats	5
3.1	Visualisation	6
3.2	Accessibilité	7
3.3	SMC	7
3.3.1	SMC quantitatif	7
3.3.2	SMC qualitatif	7
3.3.3	Q-learning	8
4	Conclusion	8

1 Introduction

Ce rapport décrit le projet réalisé dans le cadre du cours de Modélisation Probabiliste et Apprentissage par Renforcement. L'objectif du projet était d'interpréter et de simuler des chaînes de Markov (MC) et des Processus de Décision Markovien (MDP) à l'aide de Python, en utilisant des fichiers de grammaire fournis par Antlr4. En plus, nous avons implémenté plusieurs algorithmes de Model Checking sur ces processus stochastiques.

Le projet consistait en plusieurs étapes, notamment la création des classes *State*, *Action*, *Transition* et *gramDataListener* pour extraire les informations sur les processus des fichiers de grammaire, ainsi que la mise en œuvre d'une fonction permettant de se déplacer dans le graphe de manière aléatoire. Nous avons également utilisé la bibliothèque Graphviz pour visualiser le parcours de graphe.

Enfin, nous avons utilisé des méthodes SMC quantitatives et qualitatives, des techniques d'analyse de chemin ainsi que des algorithmes pour optimiser le succès de l'apprentissage par renforcement (RL). Les résultats obtenus nous ont permis d'obtenir des plus de connaissances sur les chaînes de Markov et les MDP, ainsi que sur le Model Checking et l'utilisation de Python pour s'approcher des solutions mathématiques.

Le compte rendu est organisé comme suit : la section II décrit les détails de mise en œuvre du projet, la section III présente les résultats et la section IV conclut le rapport.

Lien vers le Git que nous avons utilisé : <https://github.com/Pepitoleroii/MPARv2>

2 Mise en œuvre

Dans cette section, nous détaillons les différentes étapes de la mise en œuvre de notre projet d'interprétation et simulation de MC et MDP sur Python, en utilisant les fichiers de grammaire fournis. Nous avons divisé cette section en plusieurs sous-sections pour une meilleure organisation.

2.1 Extraction des informations sur les processus à partir des fichiers de grammaire

L'objectif était de créer ces classes pour pouvoir lire un fichier contenant les informations sur le processus afin d'en extraire les états, les actions et les transitions. Pour cela, nous avons utilisé la bibliothèque Python ANTLR4, qui permet de générer un analyseur syntaxique à partir d'une grammaire donnée.

Cela nous a permis de lire un fichier contenant les informations sur le processus et d'en extraire les différents éléments.

Nous avons créé les classes *State*, *Action* et *Transition* pour stocker les informations sur les différents éléments du processus. C'est la classe *gramDataListener* qui nous permet pour d'enregistrer les informations sur le graphe générées par l'analyseur syntaxique dans les classes correspondantes.

Concernant la structure des données. Il fallait différencier les MC ainsi que les MDP, ainsi que les processus pour lequel nous avons des récompenses. Lors de la détection d'un état sans récompense (*enterStatenoreward*), nous créons une instance de la classe *State* avec l'identifiant correspondant et une chaîne de caractère vide pour la récompense. Pour les états avec récompense (*enterStatereward*), nous créons également une instance de la classe *State*, mais cette fois-ci, nous spécifions la récompense en utilisant la valeur associée à l'identifiant.

De même, lors de la détection d'une action (`enterDefactions`), nous créons une instance de la classe *Action* avec l'identifiant correspondant. Enfin, lors de la détection d'une transition avec une action (`enterTransact`) ou sans action (`enterTransnoact`), nous créons une instance de la classe *Transition* en spécifiant le départ, l'action (le cas échéant), les états cibles et les poids associés.

En choisissant ces classes et en stockant les données dans des listes, nous avons rendu l'analyse et la manipulation de la grammaire de Markov plus facile à réaliser. Nous pouvons maintenant facilement accéder à chaque état, chaque action et chaque transition en utilisant les listes correspondantes.

2.2 Vérification

La toute première chose que nous avons fait suite à cela a été de vérifier certaines propriétés afin d'être sûr que nous nous situons dans une MC/MDP valide. Voici la liste des propriétés vérifiées :

- Chaque vérification est effectuée avec une action définie (ou bien l'action sans nom " ")
- Chaque transition part d'un état bien défini vers un autre état lui aussi bien défini.
- On vérifie que chaque état n'est pas à la fois déterministe et non déterministe.
- Chaque état à une transition.

2.3 Implémentation de la fonction pour se déplacer de manière aléatoire dans un graphe

Nous avons ensuite implémenté la fonction `"random_walk"` pour se déplacer de manière aléatoire dans le graphe. La fonction `"random_walk"` prend en entrée une liste de transitions et un état de départ. Elle commence par afficher l'état courant, puis sélectionne les transitions qui partent de cet état. Si aucune transition n'est trouvée, la fonction retourne l'état courant. Sinon, elle choisit aléatoirement une transition parmi les transitions disponibles. Ensuite, elle normalise les poids de chaque transition et sélectionne aléatoirement un état cible selon ces poids. Enfin, elle affiche l'action correspondant à la transition choisie ainsi que le prochain état et le retourne.

"Random_walk" nous permet donc de passer d'un état à un état suivant dans les MC et les MDP en tenant compte des poids de chaque transition. On a ensuite implémenté la fonction `"walk"` qui nous permet de simuler une marche aléatoire avec n transitions.

2.4 Visualisation du parcours de graphe avec Graphviz

Nous avons utilisé la bibliothèque Graphviz pour générer une visualisation du graphe et du parcours effectué avec la fonction `"walk"`. Ainsi, La fonction `visu_random_walk` permet de visualiser un processus de marche aléatoire en direct. Elle prend en entrée des informations sur les états, les actions et les transitions entre les états, ainsi qu'un entier n qui spécifie le nombre de transitions aléatoires à effectuer.

2.5 PCTL pour vérifier $P(Es)$ sur une MC

2.5.1 Principe de la fonction `"identifier_ensembles"`

On commence par identifier les états qui satisfont directement la propriété $P(Es)$, c'est-à-dire les états s pour lesquels P est vraie en un pas. Ces états seront inclus dans $S1$. Ensuite, pour tous les états restants qui ne satisfont pas directement la propriété $P(Es)$, on va chercher à déterminer si la propriété peut être satisfaite en un nombre fini de pas. Pour cela, on peut utiliser un algorithme de `model_checking` basé sur le parcours en profondeur qui permet de calculer S , l'ensemble d'états qui

satisfont la propriété en un nombre fini de pas. Enfin, les états restants qui ne satisfont pas P en un nombre fini de pas seront inclus dans S_0 .

2.5.2 Model checking avec un parcours en profondeur

Notre fonction "model_checking" vérifie si la propriété P est vraie en partant de l'état s . Elle prend en entrée un état initial, une fonction de transition qui renvoie les états accessibles depuis un état donné et une fonction P qui évalue une propriété sur un état donné. L'algorithme parcourt ensuite tous les états accessibles depuis l'état initial en explorant les états en profondeur, en utilisant une pile pour stocker les états à visiter. Lorsqu'un état satisfait la propriété, l'algorithme retourne True. Lorsqu'il n'y a plus d'états à visiter, l'algorithme retourne False si aucun état n'a satisfait la propriété.

2.5.3 Méthode du gradient conjugué

Une fois que l'on a construit nos ensemble S , S_0 et S_1 . On peut résoudre le système linéaire $Ax = b$ avec la méthode du gradient conjugué en utilisant la fonction "conj_grad".

2.6 Application de méthodes SMC quantitatives et qualitatives

2.6.1 SMC quantitative pour une MC

Le programme implémente l'algorithme de SMC quantitatif utilisant Monte Carlo pour estimer la probabilité qu'un état terminal S soit atteint en n tours ou moins. Cette méthode permet de prendre en compte les incertitudes liées à la modélisation et à l'environnement. Les paramètres de l'algorithme sont le nombre de transitions, la précision epsilon et le taux d'erreur delta.

On y calcule la borne de Chernoff-Hoeffding pour avoir le nombre minimale d'itérations nécessaires et pour chaque itération, l'algorithme utilise la fonction "etat_final" pour générer un état aléatoire. Si cet état est égal à l'état terminal S , un compteur est incrémenté de 1. L'algorithme affiche finalement le rapport entre le nombre de fois où S a été atteint et le nombre total d'itérations.

Cette méthode permet d'obtenir une estimation de la probabilité que l'état terminal S soit atteint en n tours ou moins, avec une certaine marge d'erreur.

2.6.2 SMC qualitative pour une MC

Le but de la fonction "smc_qualitative" est de déterminer si la probabilité que le modèle atteigne un état terminal S en n coups ou moins est inférieure à une certaine borne supérieure θ donnée.

Pour cela, l'algorithme utilise une borne inférieure α et une borne supérieure β , qui encadrent la probabilité cherchée. On commence par déterminer les logarithmes de LA et LB qui sont respectivement les logarithmes de la borne supérieure et de la borne inférieure. Ensuite, l'algorithme effectue une boucle de simulations. À chaque itération de la boucle, il simule un état final atteint en n coups ou moins à partir de l'état initial, en utilisant les états possibles, les actions possibles et les transitions possibles. Si l'état final simulé est l'état terminal S , alors on met à jour la somme R_m avec le logarithme de la probabilité d'atteindre l'état S en n coups ou moins. Sinon, on met à jour R_m avec le logarithme de la probabilité de ne pas atteindre S en n coups ou moins.

Si à un moment donné R_m dépasse la borne supérieure LA , alors l'algorithme nous signifie que la probabilité cherchée est supérieure à la borne supérieure θ . Si R_m est inférieur à la borne inférieure LB , alors l'algorithme renvoie 0, ce qui signifie que la probabilité cherchée est inférieure à la borne supérieure θ . Si l'algorithme a simulé un nombre maximal de simulations fixé à max ,

il renvoie "Pas fini", ce qui signifie qu'il n'a pas réussi à déterminer si la probabilité cherchée est inférieure ou supérieure à la borne supérieure θ .

2.7 Application des itérations de valeurs pour optimiser l'apprentissage par renforcement

Nous avons implémenté l'algorithme d'itération de valeurs et utilisé l'apprentissage par renforcement pour optimiser les décisions à prendre dans le modèle.

Dans notre algorithme d'itération de valeur pour un modèle de Markov à temps discret avec des gains, la fonction "transi_depart" calcule les états accessibles depuis un état donné avec une action donnée, ainsi que les probabilités associées à chaque transition. La fonction "iteration_valeur" prend en entrée l'erreur maximale ϵ , le facteur d'actualisation γ , une liste d'états $states$, une liste d'actions, et une liste de transitions.

L'algorithme calcule itérativement la valeur de chaque état en partant d'une valeur initiale de 0 pour chaque état, en utilisant l'équation de Bellman. Le processus continue jusqu'à ce que la différence entre les valeurs calculées à deux itérations successives soit inférieure à l'erreur maximale ϵ .

Une fois que les valeurs convergent, l'algorithme choisit l'action optimale pour chaque état en choisissant l'action qui maximise la somme des récompenses attendues pondérées par leur probabilité de transition vers les états accessibles suivants. Pour chaque état, l'algorithme calcule la valeur espérée de chaque action en faisant une moyenne pondérée des valeurs des états accessibles avec cette action. Ensuite, l'algorithme choisit l'action qui donne la plus grande valeur attendue pour l'état donné.

Enfin, l'algorithme retourne la liste des actions optimales pour chaque état ainsi que la liste des valeurs associées.

2.8 Application de l'algorithme Q-learning

Nous avons implémenté l'algorithme Q-learning qui nous permet d'obtenir Q . Q est une matrice de dimensions (nombre d'états) x (nombre d'actions) où chaque élément (i, j) correspond à la valeur de la qualité de l'action j dans l'état i . Cette matrice est donc mise à jour itérativement au cours de l'exécution de l'algorithme afin de permettre d'apprendre quelle est la meilleure action à prendre dans chaque état.

Dans le code, la fonction "choose_state" nous permet de changer d'état si on modifie 3 fois de suite le même élément de la matrice Q . Pour déterminer le choix de l'action, on choisit le dilemme Exploration / Exploitation epsilon greedy.

2.9 Résumé du travail effectué

Nous avons mis l'accent sur les points clés tels que l'extraction des informations à partir des fichiers de grammaire, la visualisation du parcours de graphe, l'application de méthodes SMC et d'itérations de valeurs, ainsi que l'application de l'algorithme Q-learning.

3 Résultats

Dans cette partie, nous allons montrer certains exemples de compilation de nos différents programmes.

3.1 Visualisation

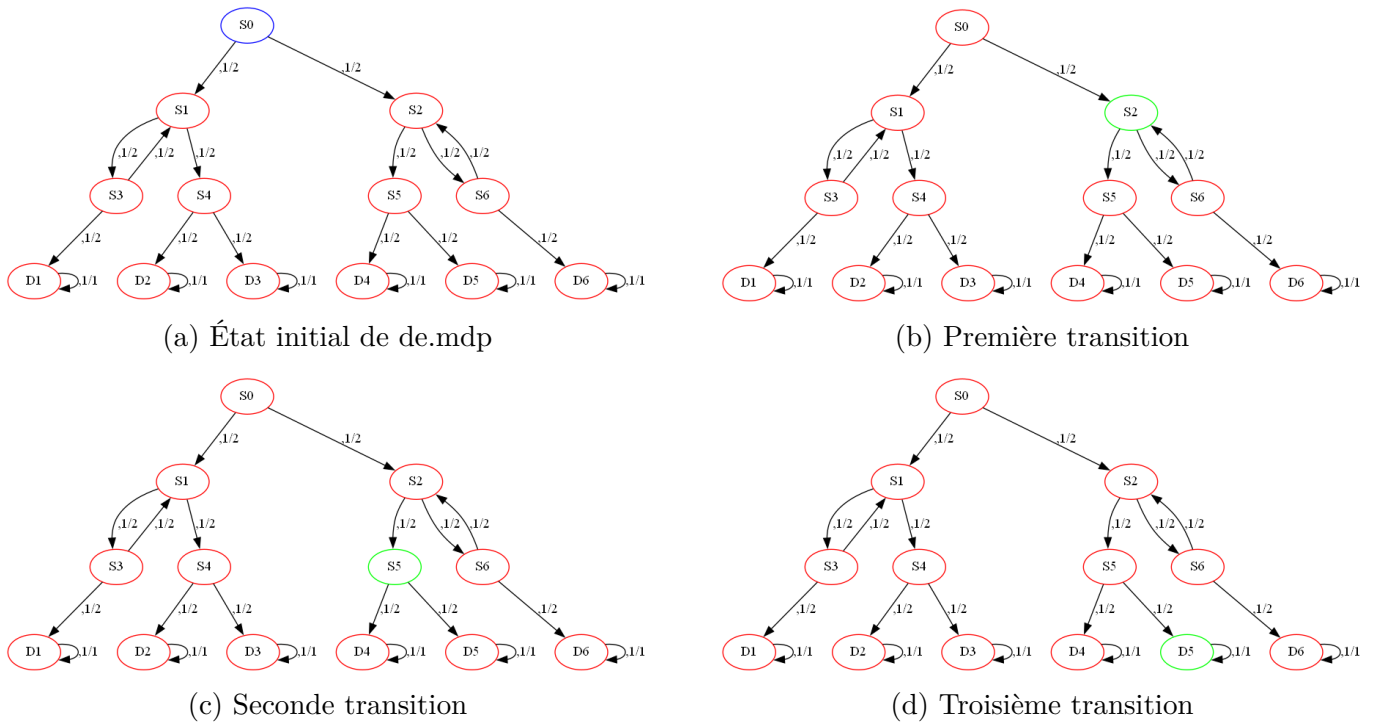


FIGURE 1 – Un exemple de parcours de de.mdp avec le programme visu_random_walk

Une limite de notre programme est que ce dernier génère autant d'images qu'il y a de transition. À notre échelle, cela n'a pas été gênant, cependant, pour des problèmes plus complexes, on aurait pu afficher le résultat sous forme de GIF.

3.2 Accessibilité

```
state = input ("quel état ?")
accessibilite(state,transitions, states)

✓ 2.2s

S0 = ['D2', 'D3', 'D4', 'D5', 'D6', 'S4', 'S5', 'S2', 'S6'] ; S1 = ['D1'] ; S? = ['S0', 'S1', 'S3']
```

FIGURE 2 – Compilation de acces.png pour l'état D1 dans l'exemple du dé

Remarque : La fonction marche également avec les MDP, il faut simplement choisir un adversaire (positionnel) et le rajouter aux arguments afin d'obtenir les listes $S_0, S_1, S_?$.

3.3 SMC

3.3.1 SMC quantitatif

```
eps = 0.01
delt = 0.1
n = 5
S = 'D1'
#de.mdp

smc_quantitatif(S,n,eps,delt,states,actions,transitions)

✓ 0.7s

0.14901198079039712
```

FIGURE 3 – Exécution de smc quantitatif pour de.mdp et l'état D1

3.3.2 SMC qualitatif

```
#de.mdp

#smc qualitatif (alpha,beta,theta,n,S, states, actions, transitions)
smc_qualitatif(0.1,0.1,0.12,5,'D1',states,actions,transitions)

Python

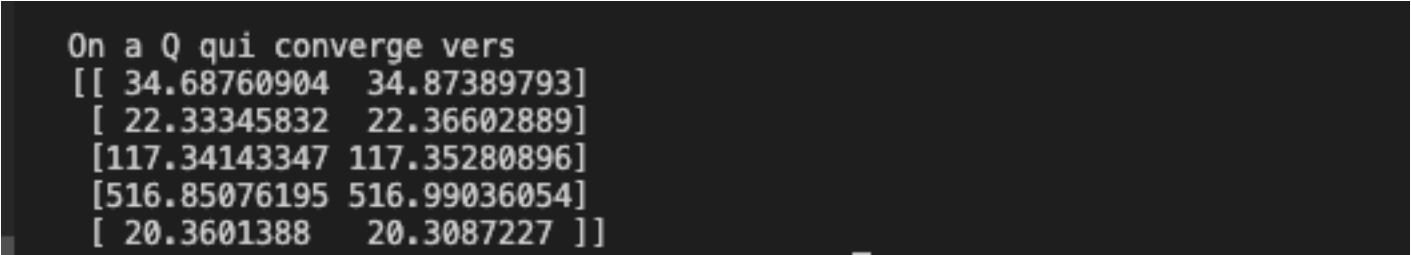
-0.9542425094393249 0.9542425094393249
plus grand que 0.12
```

FIGURE 4 – Exécution de smc qualitatif pour de.mdp et l'état D1

Remarque : Plus la valeur donnée est proche de la valeur d'équilibre, plus le programme tourne longtemps avant de converger. De plus, la valeur de précision ε impacte également grandement le temps d'exécution du programme.

3.3.3 Q-learning

On retrouve le résultat théorique du cours quand on implémente l'algorithme de Q-learning sur le fichier ex2.mdp.



```
On a Q qui converge vers
[[ 34.68760904  34.87389793]
 [ 22.33345832  22.36602889]
 [117.34143347 117.35280896]
 [516.85076195 516.99036054]
 [ 20.3601388  20.3087227 ]]
```

FIGURE 5 – Matrice Q pour l'exercice 2 du chapitre 3 traité en cours (fichier ex2.mdp)

4 Conclusion

Ce projet nous a permis de mettre en pratique un grand nombre de notions vues en cours. Il a de plus été très agréable de le voir avancer au fur et à mesure des nouveaux concepts présentés.

Les axes d'amélioration de notre projet se situent principalement dans la partie visuelle (notamment pour le parcours). Il faudrait également tester nos programmes avec des MC ou des MDP conséquentes, afin de constater si oui ou non, les fonctions ont été programmées de manière efficace. Après coup, on aurait aussi pu créer une classe MDP/MC pour pouvoir implémenter des algorithmes plus exhaustifs.