Résolution de jeux stochastiques à somme nulle à deux joueurs

Yoones MIRHOSSEINI, Marc VINCENT
Encadrant : Emmanuel HYON
Master 1 Androïde
Sorbonne Université

24 mai 2018

Table des matières

1 Introduction

Le sujet de ce projet porte sur les jeux stochastiques à somme nulle à deux joueurs. L'objectif est dans un premier temps de concevoir une modélisation informatique de ces jeux et dans un second temps d'implémenter et comparer des algorithmes permettant de résoudre ces jeux. Les algorithmes testés seront des types suivants : par apprentissage par renforcement et par l'algorithme dit de Shapley avec programmation linéaire. Le but de ces algorithmes est de trouver l'équilibre de Nash en stratégie mixte s'il existe, afin d'en déduire la ou les stratégie(s) optimale(s) pour chaque joueur. Trois jeux serviront à tester nos implémentations : pierre-papier-ciseaux, une version simplifiée du jeu de football et un jeu d'attaque-défense sur un graphe.

Nous commencerons par présenter les modèles mathématiques et les algorithmes étudiés, puis nous détaillerons notre démarche lors de la conception du logiciel, enfin nous analyserons les résultats des tests.

1.1 Descriptif général

Nous présentons ici le détail des tâches que nous avons réalisées pour ce projet. Ces tâches correspondent aux objectifs qui avaient été fixés dans le cahier des charges au début du projet.

- une bibliographie portant sur les bases théoriques des jeux stochastiques et sur les algorithmes de résolution proposés dans la littérature ;
- une phase de conception et de modélisation : nous définirons nos notations ainsi que des structures de données (c'est-à-dire un langage de description) décrivant de manière générique un jeu stochastique, ce qui permettra d'en faire une implémentation générique ;
- une définition et instanciation des jeux de tests que nous utiliserons : pierre-papier-ciseaux, jeu de football simplifié, attaque-défense sur un graphe (optionnel) ;
- une étude des *use cases* de notre logiciel, c'est-à-dire des exemples d'utilisation pour aider à prendre en main le logiciel ;
- une implémentation des algorithmes de résolution :
  1. l'algorithme dit de Shapley, avec utilisation de la programmation linéaire pour la résolution du jeu statique à chaque étape ;
  2. l'algorithme Minimax-Q-learning, basé sur l'apprentissage par renforcement ;
- la rédaction d'une documentation, du même genre que Sphinx, permettant de faciliter l'utilisation du logiciel ainsi que les modifications ultérieures par d'autres personnes ;
- une phase de tests et de comparaison : nous vérifierons que les algorithmes convergent (et à quelle vitesse) et nous nous assurerons grâce à l'équation de point fixe que la solution renvoyée est un équilibre de Nash ; en effet, la convergence n'est pas garantie par tous les algorithmes traitant les

MDP multi-agents. On fera notamment jouer deux agents avec deux différentes stratégies (trouvées par deux différentes méthodes) pour comparer le taux de victoire de chaque agent.

1.2 Déroulé du projet

Nous avons mené ce projet en plusieurs phases, en suivant de près le calendrier que nous nous étions donné en début de semestre :

— février et début mars : recherche bibliographique ;
— début mars : rédaction du cahier des charges ;
— fin mars : conception ;
— début avril : implémentation des algorithmes ;
— fin avril : tests et analyse ;
— début mai : rédaction du rapport.

2 Modèles mathématiques

Dans cette section, nous donnons une description formelle des modèles théoriques étudiés dans ce projet : les processus de décision markoviens et les notions centrales de la théorie des jeux, ainsi que l'intersection de ces deux champs d'étude, les jeux stochastiques. Nous réservons également une partie à la description des jeux-jouets sur lesquels nous effectuerons nos tests.

2.1 Les processus de décision markoviens

Dans ce projet, nous faisons usage de la notion de processus de décision markovien (MDP). Un MDP est la formalisation d'un « processus de décision séquentielle dans l'incertain » [1] . Cela permet de modéliser des situations où un agent doit prendre plusieurs décisions à la suite, sachant que les conséquences de ces décisions pour l'agent ne sont pas certaines ; il s'agit ainsi d'un modèle de décision dynamique. Plus concrètement, dans un MDP, à un instant donné, l'agent se trouve dans un certain état et doit prendre une décision ; en fonction de sa décision, il aura des probabilités différentes de passer dans chacun des autres états accessibles depuis l'état courant.

Un MDP est formellement défini par :

— un espace fini d'états S ;
— un espace fini d'actions A ;
— un axe temporel K, qui est ensemble discret fini ou infini, assimilé à un sous-ensemble de $\mathbb{N}$;
— une fonction de transition entre états $T : S \times A \times S \mapsto [0, 1]$ qui donne la probabilité de transition d'un état à un autre en fonction de l'action choisie ;
— une fonction de récompense $R : S \times A \mapsto \mathbb{R}$.

Lorsque l'on se trouve dans un état s et que l'on effectue une action a, on reçoit une récompense $R(s, a)$ et on a une probabilité $T(s, a, s')$ de passer à l'état s'. S, A, T et R peuvent dans certains cadres varier au cours du temps, c'est-à-dire selon K ; nous nous limiterons ici au cas où ils sont tous stationnaires, c'est-à-dire identiques au cours du temps.

Un MDP est un « processus stochastique contrôlé satisfaisant la propriété de Markov » [1], au sens où la probabilité de transition d'un état à un autre ne dépend que de l'état courant. On parle également d'indépendance historique.

La solution d'un MDP est une stratégie permettant de maximiser les gains obtenus. Le principe d'une stratégie, également appelée «politique» ou «règle de décision», est de fournir l'action que l'on réalisera à chaque étape ; il s'agit d'une stratégie soit mixte (c'est-à-dire probabiliste), soit pure (déterministe). Si π est une stratégie pure, $\pi(s)$ désigne l'action prise dans l'état s ; si c'est une stratégie mixte, $\pi(s, a)$ désigne la probabilité de choisir l'action a dans l'état s. L'ensemble des stratégies possibles est noté Π.

Le critère d'évaluation d'une stratégie π est l'espérance de la somme des gains obtenus en suivant cette stratégie et en partant d'un sommet initial s ; cette espérance est appelée *fonction de valeur*. L'une des

variantes les plus courantes est de pondérer les gains par un facteur d'actualisation $\gamma \in [0, 1[$ afin de prendre plus ou moins en compte la valeur des gains dans le futur. En notant r_t la récompense reçue à l'instant $t \in K$ et s_0 l'état initial, on définit ainsi la fonction de valeur pour tout $s \in S$ pour une stratégie π :

$$V^{\pi}(s) = E^{\pi}(\sum_{t=0}^{\infty} \gamma^t r_t | s^0 = s)\,.$$

Exprimer la fonction de valeur comme un cumul espéré permet d'établir le *principe d'optimalité de Bellman* [2]. Par conséquent, si π est une stratégie pure, la fonction peut également s'écrire récursivement de la manière suivante :

$$V^{\pi}(s) = R(s, \pi(s)) + \gamma \sum_{s' \in S} T(s, \pi(s), s') V^{\pi}(s')\,.$$

Si π est une stratégie mixte, alors on pose :
— $R(s, \pi(s)) = \sum_{a \in A} \pi(s, a) R(s, a)$ la récompense espérée obtenue en s en suivant π ;
— $T(s, \pi(s), s') = \sum_{a \in A} \pi(s, a) T(s, a, s')$ la probabilité de passer de s à s' en suivant π.
En utilisant ces notations, la formule ci-dessus reste valide quand π est une stratégie mixte.

Le but est de trouver la stratégie qui maximise la fonction de valeur, c'est-à-dire la stratégie optimale π^* telle que :

$$\forall s \in S \pi^*(s) = argmax_{\pi \in \Pi} V^{\pi}(s)\,.$$

On note de plus $V^*(s) = max_{\pi \in \Pi} V^{\pi}(s)$. L'équation d'optimalité associée, ou équation de point fixe, est la suivante :

$$\forall s \in S, V^*(s) = max_{\pi \in \Pi} \{R(s, \pi(s)) + \gamma \sum_{s' \in S} T(s, \pi(s), s') V^*(s')\}\,.$$

2.2 La théorie des jeux

2.2.1 Modèle de jeu

Ce projet s'inscrit également dans le large contexte de la théorie des jeux, qui est «un formalisme qui vise à étudier les interactions entre individus » [3] et qui couvre un très large champ d'applications. Dans un jeu, les participants, appelés joueurs, doivent choisir des actions à réaliser afin de maximiser une valeur appelée le gain (fonction d'utilité ou mesure de performance). Nous nous intéressons ici aux jeux non-coopératifs : les joueurs n'ont pas la possibilité de passer des accords entre eux. Lorsque chaque agent n'a qu'une décision à prendre, ce type de jeu peut être formalisé sous la forme suivante :
— un ensemble de joueurs $J = \{1, 2, ...\}$;
— pour chaque joueur $j \in J$, un ensemble d'actions possibles A_i, avec a_i l'action choisie par le joueur ;
— un ensemble des actions conjointes $A = A_1 \times A_2 \times ...$ où A_j est l'ensemble d'actions du joueur $j \in J$. On désignera par a_j l'action du joueur j et par $a = (a_1, a_2, ...)$ l'action conjointe des joueurs. De plus, a_{-j} les actions conjointes de tous les joueurs sauf j ;
— une fonction de récompense $R_j : A \mapsto \mathbb{R}$ pour chaque joueur $j \in J$ qui donne le gain associé à une action conjointe a.

Un tel jeu peut être décrit sous forme stratégique, c'est-à-dire avec une matrice des gains pour les différents joueurs en fonction des actions de chacun. Par ailleurs, nous nous plaçons dans le cadre des jeux à information complète : chaque joueur connaît les actions possibles et les gains de tous les autres joueurs.

2.2.2 Stratégies

La stratégie d'un joueur (ou règle de décision, ou politique) désigne le choix d'action(s) du joueur. Il existe des stratégies pures, c'est-à-dire déterministes (une seule action choisie) et des stratégies mixtes, c'est-à-dire probabilistes. Une stratégie mixte est une distribution de probabilité sur les stratégies pures du joueur : $\pi_j = \Delta A_j$. De même que pour les actions, on note π le vecteur des stratégies de tous les joueurs (la *stratégie conjointe*) et π_{-j} le vecteur des stratégies de tous les joueurs à l'exception de celle du joueur j.

L'étude d'un jeu vise généralement à déterminer les meilleures stratégies possibles des différents joueurs. Différents critères peuvent être utilisés pour cela. En termes de stratégies pures, une stratégie dominante $a_j \in A_j$ pour le joueur j est telle que a_i apporte au joueur un gain plus élevé que toutes ses autres actions possibles quelles que soient les actions de ses adversaires : $\forall a'_j \neq a_j \forall a_{-j} R_j(a_j, a_{-j}) \geq R_j(a'_j, a_{-j})$. Cette notion permet de définir un équilibre en stratégies dominantes : une action conjointe qui est la combinaison des stratégies dominantes de chaque joueur ; malheureusement la plupart du temps tous les joueurs n'ont pas de stratégie dominante.

L'équilibre de Nash permet de définir de manière plus pratique et plus commune les stratégies optimales d'un jeu : il s'agit d'une action conjointe dont aucun des joueurs n'a intérêt à dévier.

Définition 1 (Équilibre de Nash)**.** *Une action conjointe a^* est un équilibre de Nash si et seulement si :* $\forall j \ \forall a_j \ R_j(a^*) \geq R_j(a_j, a^*_{-j})$.

Les équilibres de Nash sont bien plus fréquents que les équilibres en stratégies dominantes, il peut même y en avoir plusieurs dans jeu. La propriété la plus intéressante des équilibres de Nash est que tout jeu en forme stratégique fini admet un équilibre de Nash en stratégies mixtes [3].

2.2.3 Jeux à somme nulle

Un type de jeu particulièrement étudié est le jeu à somme nulle : dans ce type de jeu la somme des gains de tous les joueurs s'annule : $\forall a \in A \sum_{j \in J} R_j(a) = 0$. Ce sera le cas des jeux que nous étudierons. Dans le cas des jeux à somme nulle à deux joueurs, étant donné les stratégies π_1 et π_2 des joueurs, on note $R(\pi_1, \pi_2)$ la fonction de gains, qui est la récompense du joueur 1, celle du joueur 2 étant $-R(\pi_1, \pi_2)$. Le joueur 1 veut donc maximiser la valeur de ses gains et le joueur 2 la minimiser. La valeur du jeu est alors définie de la manière suivante :

$$V^* = max_{\pi_1 \in \Pi_1} \ min_{\pi_2 \in \Pi_2} \ R(\pi_1, \pi_2) = min_{\pi_2 \in \Pi_2} \ max_{\pi_1 \in \Pi_1} \ R(\pi_1, \pi_2)$$

Cette valeur du jeu peut ainsi être déterminé par l'algorithme minimax, que nous verrons en 3.1.

2.2.4 Jeux répétés

Nous avons considéré jusqu'ici des jeux se jouant en une seule fois (une seule action de chaque joueur), mais il existe également des jeux dynamiques, qui se jouent en plusieurs tours. Il y a deux types de jeux dynamiques : les jeux en information parfaite, où les joueurs jouent chacun leur tour, et les jeux répétés, où tous les joueurs choisissent simultanément leur action. Ces jeux impliquent donc un ensemble de périodes P : on désigne par a^t l'action conjointe à une étape $t \in K$.

2.3 Les jeux stochastiques

2.3.1 Modèle

Les jeux que nous étudierons sont des jeux stochastiques. Les jeux stochastiques sont un modèle réunissant les jeux dynamiques répétés et les MDP. Ce type de jeux implique donc [4] :

- un ensemble fini d'états S ;
- un ensemble fini d'états initiaux $I \in S$, doté d'une distribution de probabilité p_I ;
- un ensemble de joueurs $J = \{1, 2\}$. Il peut y avoir plus de joueurs mais dans notre étude nous nous limiterons à 2 ;
- un ensemble d'actions $A = A_1 \times A_2$ où A_j est l'ensemble d'actions du joueur $j \in J$. On désignera par $a_{j,k}$ la k-ième action possible du joueur j, par a_j l'action du joueur j à une étape donnée et par $a = (a_1, a_2)$ l'action conjointe des joueurs à une étape donnée ;
- une fonction de transition $T : S \times A \times S \mapsto [0, 1]$ qui donne la probabilité de transition d'un état à un autre en fonction de l'action combinée de tous les joueurs. Ainsi, pour une action conjointe $a = (a_1, a_2)$, on désignera par $T(s, a, s')$ ou $T(s, a_1, a_2, s')$ la probabilité de passer de l'état s à l'état s' pour cette action ;

— une fonction de récompense $R_j : S \times A \mapsto \mathbb{R}$ pour chaque joueur $j \in J$ qui donne le gain associé à un état et une action combinée. Ainsi, dans un état s, pour une action conjointe $a = (a_1, a_2)$, on notera $R_1(s, a_1, a_2)$ ou $R_1(s, a)$ la récompense obtenue par le joueur 1 ;
— un ensemble des périodes K . A l'instant t, l'état courant sera noté s^t et l'action choisie a^t ;
— un facteur d'actualisation $\gamma \in [0, 1[$.

Ainsi, dans un jeu stochastique, chaque action conjointe mène tous les joueurs à un nouvel état où se joue l'équivalent d'un jeu statique avec des gains particuliers à chaque état. En théorie, dans un jeu stochastique, les probabilités de transition peuvent dépendre de l'historique de tous les états passés ; lorsque les probabilités de transition ne dépendent que de l'état courant, il s'agit au sens strict d'un jeu de Markov, même si cette distinction n'est pas toujours faite dans la littérature. Dans notre cadre d'étude, on considérera uniquement des jeux où les probabilités de transition ne dépendent que de l'état courant.

2.3.2 Stratégies

Dans ce type de jeux, la stratégie π_j d'un joueur j sera définie par le vecteur des stratégies $\pi_j(s)$ pour chaque état s, chacune de ces stratégies "locales" étant une distribution de probabilité sur les actions possibles dans cette état. On notera $\pi_j(s, a_j)$ la probabilité du joueur j de jouer a_j dans l'état s. Comme plus haut, π désigne la stratégie conjointe : ainsi, de même, $\pi(s, a)$ est la probabilité que les joueurs choisissent l'action conjointe a dans l'état s. Par conséquent, pour $a = (a_1, ..., a_{|J|})$, on a $\pi(s, a) = \prod_{j \in J} \pi_j(s, a_j)$.

Nous reprenons et adaptons les notations utilisées plus haut pour les MDP. On note :
— $R_j(s, \pi(s)) = \sum_{a \in A} \pi(s, a) R_j(s, a)$ la récompense espérée obtenue par j en s en suivant π ;
— $T(s, \pi(s), s') = \sum_{a \in A} \pi(s, a) T(s, a, s')$ la probabilité de passer de s à s' en suivant π.

Comme pour les MDP, le critère d'évaluation d'une stratégie π pour un joueur j est l'espérance de la somme de ses gains dans le temps en partant d'un état s. On définit de la même manière la *fonction de valeur* qui renvoie cette espérance pour tout état initial s :

$$V_j^{\pi}(s) = E\left(\sum_{t=0}^{\infty} \gamma^t R_j(s^t, \pi(s^t)) | s^0 = s\right) .$$

De même, la fonction de valeur peut s'exprimer récursivement, via une équation de Bellman [5] :

$$V_j^{\pi}(s) = R_j(s, \pi(s)) + \gamma \sum_{s' \in S} T(s, \pi(s), s') V_j^{\pi}(s') .$$

À la différence des MDP, les jeux stochastiques ne disposent pas de solutions optimales qui soient indépendantes des joueurs. Par conséquent, il faut définir la notion de stratégie optimale de manière analogue à celle d'un jeu classique. Ainsi, en notant Π_j l'ensemble des stratégies possibles du joueur j, on peut adapter la notion d'équilibre de Nash aux jeux stochastiques.

Définition 2 (Équilibre de Nash d'un jeu stochastique)**.** *Une stratégie conjointe π^* est un équilibre de Nash si et seulement si :*

$$\forall s \in S, \forall j \in J, \forall \pi_j \in \Pi_j, V_j^{\pi^*}(s) \geq V_j^{\pi_j, \pi^*_{-j}}(s) .$$

Étant donné que dans notre cadre d'étude, on considérera uniquement des jeux où les probabilités de transition sont stationnaires et ne dépendent que de l'état courant, nous chercherons également des stratégies stationnaires et dont le choix des actions dépend uniquement de l'état courant. En nous plaçant dans ce cadre, nous nous assurons que les jeux étudiés possèdent une solution grâce au théorème suivant.

Théorème 1. *[3] Tout jeu stochastique possède au moins un équilibre de Nash en stratégie stationnaire.*

2.3.3 *Competitive games*

Nous testerons nos algorithmes sur des jeux stochastiques à somme nulle à deux joueurs (même si notre code sera générique aux jeux stochastiques en général). L'avantage des jeux stochastiques à somme nulle est qu'il y existe un unique équilibre de Nash, en conséquence de l'unicité de la fonction de valeur.

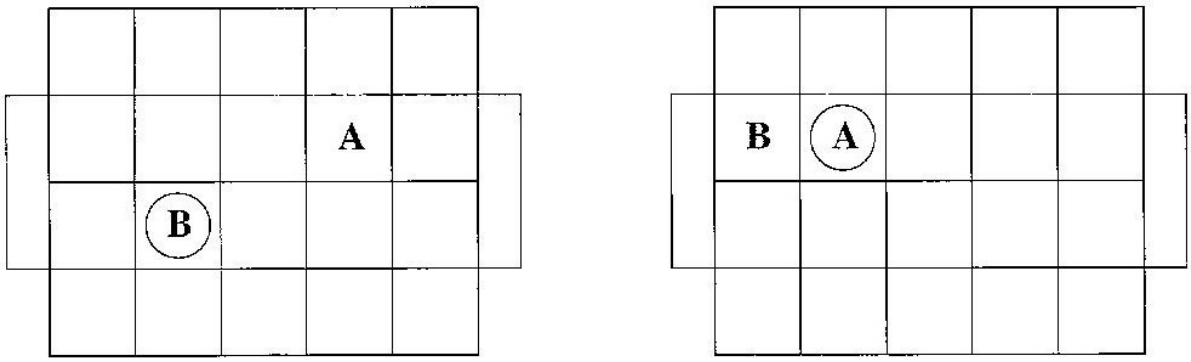

FIGURE 1 – Schéma du jeu de football simplifié

Théorème 2. *[5] Tout jeu stochastique G à somme nulle escompté à horizon infini possède une valeur unique. Cette valeur est donnée par la séquence des valeurs uniques des jeux d'état $G(s)$ $\forall s$. Chaque joueur du jeu G possède une stratégie optimale qui utilise les stratégies minimax mixtes dans chaque jeu statique associé $G(s)$ (cf. 3.1).*

L'unicité de la fonction de valeur facilitera nos tests. Ce théorème est également à la base du premier algorithme que nous testerons, l'algorithme de Shapley.

Par ailleurs, puisqu'à présent nous restreignons notre étude aux jeux à somme nulle, de même qu'en 2.2.3 nous noterons simplement $R(s, a)$ le gain du premier joueur, celui du second joueur étant $-R(s, a)$. Sans perte de généralité, nous décidons que le joueur 1 cherchera à maximiser ses gains et le joueur 2, à les minimiser.

2.4 Jeux-jouets

Les jeux-jouets sur lesquels nous testerons nos approches sont des exemples classiques des jeux stochastiques :

- pierre-papier-ciseaux est un exemple de jeu stochastique à un état, avec 3 actions disponibles, soit pratiquement le modèle le plus simple possible. C'est ainsi un cas particulier où il n'y a pas de part d'aléatoire dans l'évolution du jeu, c'est donc l'équivalent d'un jeu dynamique répété ;
- le jeu de football simplifié est présenté en [6]. Il implique deux joueurs et un ballon sur un terrain composé de 4x5 cases. Chaque agent contrôle un joueur et a au maximum 5 actions possibles (déplacement vers une case adjacente ou immobilité) : les 2 actions sont choisies simultanément mais l'ordre dans lequel elle s'exécute est aléatoire. Le ballon se déplace en fonction des actions des joueurs : lorsqu'un joueur essaye d'aller dans la case où se trouve l'autre joueur, le ballon revient systématiquement au joueur immobile. Comme l'ordre des déplacements est aléatoire, la détermination de la possession du ballon se fait de manière aléatoire dans certaines situations. Ce jeu a ceci d'intéressant qu'il y a deux états initiaux : les cases de départ des joueurs sont les mêmes dans les deux cas mais la possession du ballon est décidée aléatoirement. Le but est évidemment d'aller porter le ballon à une extrémité du terrain, ce qui rapporte une récompense de 1 au buteur et ramène à l'un des deux états initiaux ;
- l'attaque-défense dans un réseau électrique est présenté en [7] : un réseau électrique y est représenté par un graphe avec des nœuds qui créent de l'électricité, des nœuds qui en consomment et des arcs qui assurent le transport de l'électricité. Le but d'un attaquant est de minimiser la quantité d'électricité transportée, celui d'un défenseur est de la maximiser. L'action à chaque tour d'un attaquant est d'endommager un nombre limité de liens entre les nœuds du réseau, celle d'un défenseur est de le contrer en réparant ou en renforçant un nombre limité de liens. Pour chaque paire d'actions, la transition vers le nouvel état s'effectue selon une distribution de probabilité.

2.5 Commentaire bibliographique

Un descriptif de notre recherche documentaire est fourni dans le carnet de bord.

Au-delà de la recherche bibliographique, l'un de nos principaux soucis lors de la rédaction de la partie théorique de ce rapport a été d'unifier les diverses notations utilisées dans les différentes articles et livres que nous avons consultés, afin de rendre cohérente la description des différentes définitions, théorèmes et algorithmes et de mettre en avant leurs similitudes, qui n'étaient pas toujours évidentes lorsque les notations différaient beaucoup.

3 Algorithmes de résolution de jeux stochastiques à somme nulle à deux joueurs

Plusieurs algorithmes ont été proposés pour résoudre les jeux stochastiques à somme nulle. L'approche la plus fréquente est de calculer la valeur optimale du jeu pour déterminer l'équilibre de Nash et donc les stratégies optimales de chaque joueur. Pour certains de ces algorithmes, il n'existe pas de preuve qu'ils convergent ; pour d'autres, il a été prouvé qu'il existait des conditions, très restrictives, sous lesquels ils convergeaient nécessairement, mais pas forcément vers un équilibre de Nash. Nous accorderons donc une attention particulière à la convergence des algorithmes que nous testerons et à la vérification de leur solution. En revanche, à l'heure actuelle, il existe très peu d'algorithmes pour les jeux stochastiques à somme non nulle. Nous détaillons ci-dessous les algorithmes que nous étudierons.

3.1 Algorithme de Shapley avec programmation linéaire

Dans un premier temps nous allons implémenter et utiliser l'algorithme de Shapley [5] pour résoudre les jeux stochastiques à deux joueurs à somme nulle. Cet algorithme trouve l'équilibre de Nash du jeu, puis à partir de cet équilibre de Nash on est capable de calculer les stratégies optimales pour chaque joueur.

Cet algorithme fonctionne selon le principe de l'itération sur les valeurs : à chaque itération, pour chaque état de jeu possible, il faut calculer la valeur du jeu statique associé, et ce jusqu'à convergence pour tous les états, pour finalement calculer des stratégies optimales à partir de la fonction de valeur obtenue. Pour cela, on doit construire, à chaque itération n et pour chaque état s, la matrice des gains $G(s, V^n)$ du jeu statique associé à s étant donné la fonction de valeur courante V^n. Ce jeu statique (aussi appelé jeu auxiliaire) est également à somme nulle. La matrice est la suivante :

$$G(s, V^n) = \left[R(s, a_1, a_2) + \gamma \sum_{s' \in S} T(s, a_1, a_2, s') V^n(s') | a_1 \in A_1, a_2 \in A_2 \right]$$

On peut voir que ces gains statiques correspondent à la formule de la fonction de valeur présentée auparavant. L'algorithme est donc le suivant (ici pour obtenir la stratégie du joueur 1) :

Algorithme de Shapley avec programmation linéaire
Initialisation Départ avec une approximation V^0 de la valeur du jeu : pour tout s, $V^0(s)$ a une valeur quelconque. $n \leftarrow 0$ *Boucle* **répéter** pour $s \in S$ faire : Construire la matrice $G(s, V^n)$ $V^{n+1}(s) \leftarrow Valeur[G(s, V^n)]$ $n \leftarrow n + 1$ fin pour **jusqu'à** $\|V^{n+1}(s) - V^n(s)\| < \epsilon \ \forall s$ *Calcul de la politique* pour $s \in S$ faire : Construire la matrice $G(s, V^n)$ $\pi_1(s) \leftarrow Equilibre[G(s, V^n)]$ fin pour **retourner** V^n, π_1

Pour calculer la valeur du jeu et les stratégies optimales des joueurs ($Valeur$ et $Equilibre$), il faut résoudre un jeu statique à somme nulle à deux joueurs. Cela peut se faire en utilisant la technique minimax sur la matrice de gains en équivalent statique $G = G(s, V^n)$ définie plus haut : le joueur 1 essaie de maximiser les gains minimum que pourrait réaliser le joueur 2. Il nous faut donc déterminer $V(s)$ tel que :

$$V(s) = max_{\pi_1 \in \Pi_1} min_{\pi_2 \in \Pi_2} \sum_{a_1 \in A_1} \sum_{a_2 \in A_2} \pi_1(s, a_1) G[a_1, a_2] \pi_2(s, a_2)$$

$$= max_{\pi_1 \in \Pi_1} min_{a_2 \in A_2} \sum_{a_1 \in A_1} \pi_1(s, a_1) G[a_1, a_2]$$

Pour déterminer $V(s)$, l'algorithme originel utilisait l'énumération de Snow et Shapley, qui est très coûteux en temps. Pour résoudre ce problème, nous allons utiliser la programmation linéaire, qui est une technique mathématique d'optimisation de fonction objectif linéaire sous des contraintes ayant la forme d'inéquations (ou équations) linéaires. Pour calculer la valeur du jeu à chaque étape pour chaque état s, nous allons résoudre le programme linéaire correspondant à la formule ci-dessus :

$max\ V(s)$ sous contrainte que :

$$\begin{cases} \forall a_2 \in A_2, V(s) \leq \sum_{a_1 \in A_1} \pi_1(s, a_1) G[a_1, a_2] \\ \sum_{a_1 \in A_1} \pi_1(a_1) = 1 \\ \forall a_1 \in A_1, \pi_1(a_1) > 0 \\ V(s) \in \mathbb{R} \end{cases}$$

Ainsi, dans l'algorithme, $Valeur$ renvoie le $V(s)$ trouvé par résolution du programme linéaire et $Equilibre$ renvoie le π_1 trouvé.

3.2 Apprentissage par renforcement

3.2.1 Principe général

L'apprentissage par renforcement est une forme d'apprentissage non supervisé : il permet ainsi de résoudre des MDP dans lesquels les fonctions de transition et de récompense ne sont pas connues a priori. Dans notre cas, étant donné que l'environnement est complètement décrit, cette technique va simplement nous offrir une autre méthode de calcul de la politique optimale.

L'apprentissage par renforcement est basé sur l'évaluation du bénéfice à tirer des actions que peut prendre un agent dans différentes situations, dans le but de renforcer la tendance à exécuter les actions jugées bénéfiques. Pour cela, lors de l'apprentissage, l'agent teste un grand nombre de fois les différentes actions dans les différents états possibles et essaie d'en déduire celles qui lui rapportent le plus en fonction des récompenses qu'il a obtenues lors de ces tests.

Le défi dans cette approche est d'évaluer les bénéfices *à long terme* des actions. il faut donc prendre en compte la durée qui sépare l'action de sa récompense et la fréquence à laquelle cette action a été testée.

C'est donc sur la base d'une "*interaction itérée du système apprenant avec l'environnement*" [8] que l'on peut déduire une politique que l'on va améliorer au fur et à mesure. Cependant, en pratique, lors de chaque itération, la plupart des algorithmes n'essaient pas de calculer une politique mais une fonction de valeur associée au MDP ; la politique trouvée est déduite de l'approximation de la fonction de valeur obtenue à la fin de l'algorithme.

3.2.2 Le Q-learning

L'une des méthodes d'apprentissage par renforcement les plus répandues pour les MDP (pour un seul agent, donc) est le Q-learning, qui consiste à apprendre la valeur des actions selon les états, ce qui permet de calculer les utilités et la politique optimale dynamiquement [9].

En Q-learning, on associe à chaque couple état-action d'un agent une Q-valeur $Q(s, a)$ qui correspond à « la récompense espérée obtenue en effectuant l'action a dans l'état s et en suivant une politique optimale à partir de l'état suivant » [10], à savoir :

$$Q(s, a) = R(s, a) + \gamma \sum_{s' \in S} T(s, a, s')V(s')$$

La stratégie et la fonction de valeur optimales sont alors déduites de la manière suivante :

$$V^*(s) = max_{a \in A} Q(s, a)$$

$$\pi^*(s) = argmax_{a \in A} Q(s, a)$$

Ces Q-valeurs sont évaluées dynamiquement par l'expérience de l'agent dans l'environnement. Ainsi, la mise à jour de $Q(s, a)$ au moment d'effectuer l'action a pour passer de l'état s à l'état s'se fait de la manière suivante :

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha[R(s, a) + \gamma max_{a' \in A} Q(s', a')]$$

À chaque itération, le taux d'apprentissage $\alpha \in]0, 1]$ décroît, ce qui fait que l'algorithme apprend de moins en moins et les valeurs des $Q(s, a)$ finissent par se stabiliser.

Le Q-learning est un algorithme d'apprentissage mono-agent, qui peut être adapté pour un environnement multi-agent mais sans prendre en compte explicitement la présence des autres agents : en effet, dans ce cas, l'autre agent est considéré comme une partie de l'environnement. Les algorithmes de Q-learning multi-agent comme celui que nous étudierons convergent sous certaines conditions, qui concernent notamment la probabilité d'exploration [11].

3.2.3 L'algorithme Minimax Q-learning

L'algorithme d'apprentissage par renforcement que nous allons implémenter, proposé par Littman [6] et connu sous le nom de « Minimax Q-learning », est une extension de l'algorithme de Q-learning standard, adaptée aux jeux stochastiques à deux joueurs à somme nulle. Il résout le problème de la convergence prenant en compte de manière plus explicite la présence de l'autre agent via la technique du minimax de la théorie des jeux. À chaque itération, on résout le jeu statique équivalent à l'état courant s en utilisant comme matrice de gain $Q[s]$, qui est indicée par les actions des joueurs, c'est-à-dire qu'on a : $Q[s] = [Q(s, a_1, a_2) \mid a_1 \in$

$A_1, a_2 \in A_2$]. L'algorithme nécessite de prendre en paramètre la stratégie de l'adversaire pour déterminer les récompenses et les transitions ; on peut pour commencer lui fournir la stratégie aléatoire, où toutes les actions sont équiprobables.

L'algorithme est le suivant (ici pour obtenir la stratégie du joueur 1, qui cherche à maximiser). Il prend plusieurs paramètres :

- la stratégie π_2 de l'adversaire ;
- $p_{explore}$, qui représente la probabilité de choisir la prochaine action non pas selon la politique courante mais aléatoirement, afin de diversifier les tests en tentant des actions qui pourraient se révéler meilleures que celles "recommandées" par la politique courante (ce que l'on appelle le dilemme exploration/exploitation) ;
- d qui est le ratio selon lequel α diminue à chaque itération. α est le taux d'apprentissage : il est initialisé à 1 et décroît progressivement ; lorsqu'il approche de 0, l'algorithme n'apprend plus car il ne tient plus compte des récompenses qu'il reçoit. Le ratio d et le nombre d'itérations n doivent être choisis de manière à ce que α approche de 0 à la fin de la boucle.

Algorithme Minimax Q-learning
Initialisation
Pour $s \in S$, $a_1 \in A_1$, $a_2 \in A_2$:
$Q[s][a_1, a_2] \leftarrow 1$
Pour $s \in S$:
$V(s) \leftarrow 1$
Pour $s \in S$, $a_1 \in A_1$:
$\pi_1(s, a_1) \leftarrow 1/\|A\|$
$\alpha \leftarrow 1$
s : état initial décidé aléatoirement
Boucle
répéter n **fois**
Choix de l'action
Avec une probabilité $p_{explore}$, choisir l'action a_1 aléatoirement uniformément
Sinon, étant donné l'état courant s, choisir l'action a_1 avec une probabilité $\pi_1(s, a_1)$
Choisir l'action a_2 avec une probabilité $\pi_2(s, a_2)$
Apprentissage
Choisir l'état suivant s' avec une probabilité $T(s, a_1, a_2, s')$
$Q[s][a_1, a_2] \leftarrow (1 - \alpha)Q[s][a_1, a_2] + \alpha(R(s, a_1, a_2) + \gamma V(s'))$
$\pi_1(s) \leftarrow Equilibre(Q[s])$
$V(s) \leftarrow Valeur(Q[s])$
$\alpha \leftarrow \alpha * d$
$s \leftarrow s'$
retourner V, π_1

4 Modélisation et conception

4.1 Conception

4.1.1 Objectifs

Dans cette partie, nous exposons et justifions nos choix en matière d'implémentation. Le but du logiciel que nous codons est double : il doit répondre à nos besoins, que nous détaillons ci-après, et doit pouvoir être réutilisé et amélioré par d'autres personnes travaillant sur les jeux stochastiques.

L'objectif principal de ce projet est de tester et comparer des algorithmes de résolution de jeux stochastiques à somme nulle à deux joueurs. Pour cela, nous avons besoin d'implémenter un logiciel générique permettant d'appliquer ces algorithmes à plusieurs jeux sans avoir à les recoder. Pour effectuer des tests sur

ces jeux, il faut donner une représentation informatique des jeux stochastiques et coder nos algorithmes de manière à ce qu'ils s'appliquent à celle-ci. Bien que notre projet se limite aux jeux à somme nulle à deux joueurs, nous nous efforçons de donner une représentation générique des jeux stochastiques, en vue d'extensions ultérieures, ce qui concerne principalement le nombre de joueurs. Il s'agit donc en quelque sorte de définir un langage de description ou *DSL* (*Domain Specific Language*) pour les jeux stochastiques en général. Nous mettons en particulier l'accent sur l'utilisabilité de ce logiciel, la façon dont l'utilisateur devra entrer un jeu et lancer un algorithme dessus.

4.1.2 Problèmes rencontrés et solutions apportées

Au cours de la conception de notre implémentation, il a fallu que nous apportions des solutions à plusieurs problèmes, quitte à adapter par moments la modélisation mathématique. Les principaux problèmes concernaient :

1. la représentation externe, c'est-à-dire la manière dont l'utilisateur entrera les données d'un jeu ;
2. la représentation interne, à savoir les structures de données nécessaires pour contenir toutes les informations qui définissent un jeu ;
3. la modélisation des états, des actions et des joueurs, et la manière de les "identifier" ;
4. la gestion des actions impossibles dans certains états ;
5. le fait que dans la plupart des jeux envisageables, pour la grande majorité des paires d'états (s, s'), il n'y a pas de transition possible, c'est-à-dire : $\forall a\ T(s, a, s') = 0$.

Les deux derniers points peuvent être illustrés par l'exemple du jeu de football simplifié, qui nous a en grande partie guidés lors de cette phase de conception.

Premières tentatives de représentation abandonnées. Tout d'abord, il a fallu déterminer les grandes lignes de ces représentations.

Au niveau de la représentation interne, nous avons tout d'abord pensé intuitivement à des représentations matricielles. Ainsi, la première approche qui a été proposée était de stocker pour chaque paire d'états (s, s') une matrice carrée M avec les actions de chaque joueur en entrée telle que $M[a_1, a_2] = T(s, (a_1, a_2), s')$; et pour chaque état, stocker une matrice similaire pour les récompenses immédiates. Cette approche a très vite été écartée en raison du fait que ces matrices auraient été très largement vides, utilisant ainsi de l'espace mémoire inutile.

Une seconde approche se proposait de stocker pour chaque état une matrice des actions contenant pour chaque action combinée la récompense immédiate associée et une distribution de probabilité sur l'état suivant, celle-ci étant limitée aux états accessibles. Cette solution, nettement plus optimisée, semblait néanmoins trop complexe, trop peu générique, peu propice aux modifications ultérieures ; elle posait également la question de la modélisation des états et de leur identification.

Modélisation des éléments du jeu. En effet, la modélisation des actions et des joueurs peut se limiter à un simple identifiant (chaîne de caractères ou entier) alors que les états sont en général assez nombreux (760 dans le jeu de football) : si l'on modélisait les états par un identifiant, il ne serait pas trivial de déterminer les identifiants des états accessibles depuis un état donné. Par conséquent, nous avons estimé que les états complexes devraient pouvoir être modélisés sous forme de tuples pour faciliter la détermination des états accessibles : pour le jeu de football, un état serait donc sous la forme *(position du joueur 1, position du joueur 2, joueur en possession du ballon)*.

Il nous est alors apparu que dans les exemples du jeu de football et de l'attaque-défense sur un réseau électrique, et par extension dans les autres jeux envisageables, il était nécessaire que les états accessibles soient générés automatiquement plutôt que d'être rentrés à la main à l'utilisateur : de même pour les distributions de probabilité sur ceux-ci et pour les récompenses immédiates. Cela nous amène au point suivant sur la représentation externe.

Représentation externe par classe abstraite. En tentant de formaliser dans un premier temps le jeu de football d'après la modélisation mathématique, nous avons constaté que la manière la plus simple pour l'utilisateur d'entrer les transitions et les récompenses dans le logiciel était d'écrire les fonctions associées ; par ailleurs, concernant la représentation interne, il se trouve que c'est également la solution la moins coûteuse en mémoire et qu'elle fournit en plus l'API nécessaire aux algorithmes. Cette approche peut de plus être utilisée pour toutes les autres données d'un jeu : donner la liste des états, celle des actions, celle des joueurs, le facteur d'actualisation et la distribution de probabilité sur les états initiaux. Il n'y a donc *in fine* aucun besoin d'utiliser des matrices pour la représentation interne ou externe.

Par conséquent, l'implémentation la plus générique est de définir une classe abstraite *StochasticGames* contenant toutes ces méthodes, que l'utilisateur devrait implémenter pour obtenir une classe définissant un jeu particulier. Le fait de modéliser un jeu par une classe permet en outre de définir des attributs propres au jeu et utilisables dans différentes méthodes (notamment le graphe dans l'exemple de l'attaque-défense dans un réseau électrique).

L'inconvénient ici du langage choisi pour l'implémentation, Python, est qu'il ne permet pas (en tout cas de manière directe) de vérifier que l'utilisateur implémente les méthodes avec la bonne signature : le mieux que l'on puisse faire est de fournir la documentation la plus claire possible.

La fonction de transition. Cependant, la signature de la fonction de transition $T : S \times A \times S \mapsto [0, 1]$ pose problème. Certes, le fait de coder directement la fonction de transition plutôt que de donner une représentation matricielle des transitions permet d'éviter les problèmes liés aux transitions nulles entre la plupart des états. En revanche, l'exemple du jeu de football nous a appris qu'il était peu évident de coder $T(s, a, s')$ en prenant en paramètre l'état suivant s'. Mais le principal problème se pose pour les algorithmes basés sur la simulation (notamment Q-learning) et pour les tests : en effet, si T avait une telle signature, alors étant donné un état actuel et une action conjointe, la détermination de l'état suivant nécessiterait de parcourir l'ensemble des états pour trouver la probabilité de transition vers chacun d'entre eux.

Afin de simplifier l'écriture de la fonction et d'optimiser le calcul d'une transition, nous proposons que la fonction de transition prenne uniquement en entrée l'état courant et l'action et qu'elle renvoie une distribution de probabilité sur l'état suivant, sous la forme d'un dictionnaire.

Les actions impossibles. La question de la gestion des actions impossibles dans certains états s'est également posée. Dans un premier temps nous avions pensé effectuer ces vérifications dans les fonctions de transition et de récompense, notamment en attribuant une récompense équivalente à $-\infty$; cependant, cette solution aurait pu poser problème sur certains algorithmes. La solution retenue, plus naturelle pour les utilisateurs, consiste à définir les actions possibles selon l'état dans la fonction renvoyant les actions : celle-ci prend ainsi en paramètres le joueur et l'état courant et renvoie les actions possibles pour ce joueur dans cet état.

Généricité et spécificité. Par ailleurs, nous avions dans un premier temps mis dans l'interface une méthode donnant le nombre de joueurs n : on attribuait automatiquement aux joueurs un identifiant dans $[1, n-1]$. Cependant, pour permettre une implémentation plus libre et par cohérence avec les fonctions pour les actions et les états, nous l'avons remplacée par une fonction devant renvoyer la liste des joueurs (entiers ou chaînes de caractères). Par conséquent, la donnée des actions ou des récompenses de l'ensemble des joueurs à un temps donné est représentée par un dictionnaire associant une valeur à chaque joueur, là où auparavant on se contentait d'un tuple.

À l'inverse, dans notre cadre d'étude, certaines simplifications peuvent être faites par rapport à l'implémentation générique décrite jusqu'ici : il n'y a que deux joueurs (0 et 1) et la seule fonction de récompense dont nous ayons besoin est celle qui donne la récompense du joueur 0, la récompense de l'autre joueur pouvant être déduite puisque nous étudions des jeux à somme nulle. Nous mettons en place ces simplifications en créant une sous classe *NullSum2PlayerStochasticGame* de notre classe *StochasticGame*.

Néanmoins un problème se pose dans le cadre générique pour la fonction de récompense : la cohérence avec le reste de l'API voudrait que cette fonction prenne un joueur en paramètre et renvoie la récompense

de celui-ci, mais pour permettre l'optimisation dans les jeux à somme nulle il faut introduire une fonction rendant les récompenses de l'ensemble des joueurs.

Nous avons donc créé deux fonctions *_reward(player, state, actions)* et *rewards(state, actions)* : la première renvoie la récompense d'un joueur, la deuxième celles de tous les joueurs sous forme de liste. Par défaut, *rewards* fait simplement appel à *_reward* pour tous les joueurs ; seule *rewards* est utilisée dans les algorithmes, *_reward* est donc l'équivalent d'une fonction privée. *_reward* n'est pas implémentée de base dans la classe mère. L'utilisateur peut donc implémenter l'une ou l'autre au choix dans son propre jeu.

Représentation des stratégies et de la valeur du jeu. Enfin, concernant l'implémentation des algorithmes, nous avons décidé de stocker π et V sous forme de dictionnaires ; l'avantage par rapport à une matrice est que le dictionnaire permet d'utiliser n'importe quels identifiants pour les joueurs, les actions et les états (entier, chaîne de caractères ou tuple) et qu'il évite les "creux" qui auraient été présents dans une matrice.

4.2 Spécifications et implémentation

4.2.1 Choix d'implémentation

Nous avons décidé que le langage utilisé dans le projet serait Python parce qu'il nous est familier, qu'il permet la programmation objet, qu'il est portable et largement répandu et qu'il dispose de nombreuses bibliothèques qui facilitent l'implémentation. La version utilisée est Python 3.6.

Pour résoudre les programmes linéaires, nous avons utilisé Gurobi, un solveur d'optimisation pour la programmation mathématique comme la programmation linéaire, quadratique, etc. Gurobi possède plusieurs implémentations en différents langages de programmation. Nous avons utilisé l'interface Python de Gurobi.

Concernant les algorithmes, quelques détails d'implémentation méritent d'être mentionnés. Tout d'abord nous avons fait en sorte qu'ils puissent être exécutés en choisissant le rôle des joueurs (lequel veut maximiser ses gains, lequel veut les minimiser) afin de pouvoir obtenir la stratégie de chacun. De plus, nous avons implémenté la résolution du maximin par programmation linéaire séparément des algorithmes, pour pouvoir le réutiliser dans chacun. Enfin, nous avons pris soin de mettre en paramètre de l'algorithme Minimax-Q-learning la stratégie du joueur adverse.

Par ailleurs, nous avons également implémenté une classe permettant de mener des simulations faisant s'affronter deux stratégies et de comparer leurs scores.

4.2.2 Manuel utilisateur

Implémentation du jeu Pour exécuter les algorithmes sur un jeu donné, l'utilisateur doit d'abord créer une classe correspondant à ce jeu et qui hérite de la classe abstraite *StochasticGames*.

Le joueur doit implémenter les méthodes *states()* et *players()* qui doivent renvoyer respectivement une liste des états et une liste des joueurs, ainsi que *actions(player, state)* qui doit renvoyer la liste des actions possibles d'un joueur dans un état. Le choix du type des identifiants des états, des joueurs et des actions est libre : en principe il s'agira de chaînes de caractères ou d'entiers, mais il est permis et recommandé d'utiliser des tuples pour les états lorsque ceux-ci sont complexes.

La méthode *transition(state, action)* donne une distribution de probabilité sur les états accessibles depuis un état donné en fonction d'une action conjointe. Comme expliqué plus haut, les récompenses peuvent être implémentées soit dans *_reward(player, state, actions)* soit dans *rewards(state, actions)*. Enfin, la méthode *gamma* renvoie le facteur d'actualisation du jeu. En créant cette classe, l'utilisateur pourra par exemple instancier différentes variantes de son jeu.

L'utilisateur pourra aussi passer par *NullSum2PlayerStochasticGame*, où il n'aura pas à préciser *players()* et où il devra implémenter *player0_reward(state, actions)* pour renseigner les récompenses du joueur qui cherche à maximiser ses gains.

Utilisation des algorithmes Étant donné une instance *jeu* de son jeu, l'utilisateur pourra ensuite exécuter les algorithmes. Pour utiliser l'algorithme de Shapley, il devra faire appel à *shapley(jeu, epsilon, playerA, playerB)* où *playerA* est le joueur dont on veut obtenir la stratégie et *epsilon* est l'écart en-dessous duquel on considère que la fonction de valeur converge. Pour utiliser l'algorithme Minimax-Q-learning, il devra faire appel à *MinimaxQ(jeu, explor, decay, display, iteration, playerA, playerB, policyplayerB)* où *playerA* est le joueur dont on veut obtenir la stratégie, *policyplayerB* la stratégie de son adversaire, *iteration* le nombre d'itérations ; le numéro de l'itération courante sera affiché toutes les *display* itérations et les paramètres *explor, decay* sont expliqués plus haut.

Pour lancer une simulation entre deux stratégies, l'utilisateur créera une classe avec *Test(jeu, nombre_itérations, stratégies)* où *stratégies* est un dictionnaire donnant la stratégie de chaque joueur. La simulation est exécutée par la méthode *simulation()* ; les taux de victoires peuvent ensuite être obtenus avec la méthode *get_winners()*.

Illustration du code Avec l'architecture présentée, nous avons implémenté les trois jeux présentés. Nous donnons en annexe le code des classes *StochasticGames*, *NullSum2PlayerStochasticGame* et de l'exemple le plus intéressant, le jeu de football simplifié.

5 Tests et analyse des résultats

5.1 Pierre-papier-ciseaux

Nous vérifions d'abord que nos algorithmes fonctionnent sur le cas le plus simple, pierre-papier-ciseaux. Nous exécutons les deux algorithmes, avec les résultats suivants :

```
Algorithme de Shapley
Paramètres : epsilon = 0
Nombre d'itérations : 278
Temps d'exécution : 0.318 s
Valeur du jeu : 0
Politique optimale : {'pierre': 0.3333333333333337,
'ciseaux': 0.3333333333333333, 'papier': 0.3333333333333337}

Minimax-Q-learning
Stratégie de l'adversaire : aléatoire
Paramètres : nombre d'itérations = 4350, explor = 0.3,
decay = 0.01 ** (1. / ( 2 *10**4))
Taux d'apprentissage à la fin de l'exécution : 0.367
Temps d'exécution : 4.172 s
Valeur du jeu : 0
Politique optimale : {'pierre': 0.3333333333333426,
'ciseaux': 0.33333333333338067, 'papier': 0.3333333333327675}
```

On constate que les deux convergent bien. On voit que l'algorithme de Shapley (AS) est beaucoup plus rapide et que Minimax-Q-learning (MQL) met plus longtemps à converger, mais c'est un peu la nature de l'apprentissage par renforcement ; par ailleurs si l'on change les paramètres comme le taux d'apprentissage ou que l'on initialise la valeur du jeu à 0.1 au lieu de 1 par exemple, on peut réduire le temps d'exécution, voire converger aussi vite que AS.

On peut voir aussi que la somme des probabilités des actions dans les stratégies mixtes calculées par les deux algorithmes est égale à 1 à 10^{-12} près même si les probabilités ne sont pas parfaitement exactement égales.

Nous organisons alors un "match" entre les deux stratégies trouvées :

```
Test sur 10 000 000 de parties : AS versus MQL
Récompenses totales : {0: 0.899, 1: -0.899}
Taux de victoire : {0: 0.500015969671719, 1: 0.499984030328281}
```

On vérifie expérimentalement que le choix équiprobable entre les différentes actions est la stratégie optimale : sur 10^7 parties le pourcentage de victoires est le même à 10^{-4} près.

5.2 Jeu de football simplifié

Nous passons maintenant à l'exemple plus intéressant et plus complexe du jeu de football simplifié, qui implique plus d'exécutions des algorithmes et plus de tests.

5.2.1 Première phase d'apprentissage

Nous commençons par exécuter les algorithmes. Étant donné que ce jeu comporte 760 états, nous n'affichons pas ici la stratégie et la fonction de valeur obtenues. Nous utilisons à la place des mesures de distance pour comparer les fonctions de valeur : étant donné deux fonctions de valeur V_1 et V_2 on mesure le plus grand écart entre les valeurs d'un même état, et la somme des écarts sur tous les états, à savoir : $max\_ecarts(V_1, V_2) = max_{s \in S} |V_1(s) - V_2(s)|$ et $somme\_ecarts(V_1, V_2) = \sum_{s \in S} |V_1(s) - V_2(s)|$. Nous utiliserons ces mesures pour deux cas :

- comparer les fonctions de valeur des avant-dernière et dernière itérations ("écarts entre les fonctions de valeur successives"), afin de vérifier la convergence de l'algorithme ;
- comparer les fonctions de valeur obtenues par les deux algorithmes.

Pour comparer les stratégies, nous organisons des simulations entre deux stratégies.

Nous commençons par exécuter l'algorithme de Shapley.

```
Algorithme de Shapley
Paramètres : epsilon = 10 ** -13
Nombre d'itérations :  257
Temps d'exécution : 348.155 s
À la fin de l'algorithme :
- max_ecarts : 8.371e-14
- somme_ecarts : 6.350e-11
```

On voit que AS converge assez rapidement. Il apparaît également que quand on exécute l'algorithme pour le joueur adverse, la fonction de valeur trouvée est bien inverse : par exemple quand la valeur de jeu pour l'état s est 1 pour le joueur 0, elle est -1 pour le joueur 1.

Nous exécutons maintenant MQL en l'opposant à une stratégie aléatoire (MQL1).

```
Minimax-Q-learning 1
Stratégie de l'adversaire : aléatoire
Paramètres : nombre d'itérations = 10 ** 6, explor = 0.3,
decay = 0.01 ** (1. / ( 2 *10**4))
Taux d'apprentissage à la fin de l'exécution : 0.0099
Temps d'exécution : 1467.812 s
À la fin de l'algorithme :
- max_ecarts : 0.218
- somme_ecarts : 21.574
Comparaison avec AS :
- max_ecarts : 0.389
- somme_ecarts : 53.832
```

En exécutant MQL avec un million d'itérations, on voit que l'exécution est bien plus lente que pour AS, et pourtant l'algorithme est loin d'avoir convergé entièrement, avec une somme des écarts entre fonctions de valeur successives de plus de 21. L'écart avec la fonction de valeur trouvée par AG est également non-négligeable ; en observant les stratégies trouvées par AG et MQL, on constate effectivement qu'elles diffèrent pour la plupart des états.

À présent, nous faisons apprendre MQL face à la stratégie obtenue avec AS (MQL2).

```
Minimax-Q-learning 2
Stratégie de l'adversaire : stratégie obtenue par AS
Paramètres : nombre d'itérations = 10 ** 6, explor = 0.3,
decay = 0.01 ** (1. / ( 2 *10**4))
Taux d'apprentissage à la fin de l'exécution : 0.0099
Temps d'exécution : 1480.094 s
À la fin de l'algorithme :
- max_ecarts : 1.929
- somme_ecarts : 275.418
Comparaison avec AS :
- max_ecarts : 1.899
- somme_ecarts : 365.621
```

On constate clairement que les écarts entre les fonctions de valeur successives sont beaucoup plus élevés lors de l'apprentissage contre la stratégie trouvée par AS que lors de l'apprentissage contre la stratégie aléatoire. Cela montre que si l'on veut que MQL converge vers l'équilibre de Nash, il faut apprendre les valeurs de jeu contre la stratégie aléatoire. Cependant même en un million d'itérations contre la stratégie aléatoire MQL n'a pas convergé ; nous lancerons donc une seconde phase d'apprentissage avec plus d'itérations pour essayer de converger complètement.

En regardant plus en détail la stratégie et la fonction de valeur obtenues contre la stratégie de AS, on observe que de nombreux états n'ont jamais été modifiés : ils ont la même valeur de jeu et les mêmes probabilités d'action qu'à l'initialisation de l'algorithme (c'est-à-dire 1 et une probabilité uniforme). L'explication est la suivante : la stratégie trouvée par AS ne joue jamais certaines actions, donc on ne rentre jamais dans ces états lors de l'algorithme et par conséquent leur valeur ne change jamais. Ainsi, MQL apprend la valeur et les probabilités des états nécessaires uniquement, ceux dans lesquels il doit répondre à une action effectuée par l'adversaire.

Cela explique aussi pourquoi les écarts entre les fonctions de valeur successives de MQL et ceux entre les valeurs de MQL et de AS sont aussi importants. En effet, de nombreux états conservent la valeur 1 (valeur de l'initialisation) qui n'est pas correcte. Cependant, pour les états dont la valeur de jeu a été modifiée, les écarts entre les valeurs de jeu pour ces états trouvées par AS et par MQL est faible ; de même les stratégies obtenues sont semblables pour ces états. Il est donc possible que cela ne pose pas de problème pour jouer contre la stratégie trouvée par AS : en effet, même si l'on n'a pas du tout convergé vers un équilibre de Nash, la stratégie trouvée par MQL contre AS pourrait se montrer efficace contre la stratégie de AS, ce que nous testerons lors de la phase de comparaisons.

5.2.2 Première phase de comparaisons

Nous réalisons à présent les simulations grâce à la classe *Test* décrite plus haut.

```
Test sur 1 000 000 de parties : MQL1 versus AS
Récompenses totales : {0: -0.0004, 1: 0.0004}
Taux de victoire : {0: 0.4500314912190764, 1: 0.5499685087809236}
Nombre de buts marqués : 68273

Test sur 1 000 000 de parties : MQL2 versus AS
Récompenses totales : {0: -0.746, 1: 0.746}
```

```
Taux de victoire : {0: 0.47818932645602263, 1: 0.5218106735439774}
Nombre de buts marqués : 63203
```

On constate que la stratégie de AS est plus efficace que celles obtenues par MQL et contre la stratégie aléatoire et contre AS : 55% et 52% de victoires respectivement. Nous allons tenter un apprentissage plus long pour MQL. Cependant, dans les deux cas, sans même que MQL ait convergé, les stratégies obtenues restent raisonnablement efficaces contre la stratégie de AS.

```
Test sur 1 000 000 de parties : AS versus aléatoire
Récompenses totales : {0: 0.451, 1: -0.451}
Taux de victoire : {0: 0.9889125840723336, 1: 0.011087415927666431}
Nombre de buts marqués : 103631

Test sur 1 000 000 de parties : MQL1 versus aléatoire
Récompenses totales : {0: 0.447, 1: -0.447}
Taux de victoire : {0: 0.9877311116183519, 1: 0.012268888381648026}
Nombre de buts marqués : 105144
```

On vérifie que la stratégie obtenue par AS et celle obtenue par MQL contre la stratégie aléatoire sortent largement gagnantes d'une simulation face à la stratégie aléatoire avec 99% de victoires. Encore une fois, MQL se montre efficace sans avoir convergé complètement.

```
Test sur 1 000 000 de parties : MQL2 versus aléatoire
Récompenses totales : {0: 0.358, 1: -0.358}
Taux de victoire : {0: 0.8997145708932355, 1: 0.1002854291067645}
Nombre de buts marqués : 57107
```

La stratégie obtenue par MQL contre AS sort également gagnante d'une simulation face à la stratégie aléatoire, mais avec un taux de victoires clairement moindre, 90%.

```
Test sur 1 000 000 de parties : AS versus AS
Récompenses totales : {0: 0.354, 1: -0.354}
Taux de victoire : {0: 0.500518276050191, 1: 0.49948172394980905}
Nombre de buts marqués : 73320
```

On vérifie expérimentalement que la stratégie obtenue par AS est la stratégie optimale, car en la faisant s'affronter elle-même on obtient un taux de victoires parfaitement équilibré.

5.2.3 Seconde phase d'apprentissage

Nous lançons donc une nouvelle phase d'apprentissage pour MQL, cette fois-ci avec 20 millions d'itérations, dans l'espoir que l'algorithme converge. L'exécution dure 8 heures.

```
Minimax-Q-learning 1
Stratégie de l'adversaire : aléatoire
Paramètres : nombre d'itérations = 20 * 10 ** 6, explor = 0.3,
decay = 0.01 ** (1. / ( 2 *10**4))
Taux d'apprentissage à la fin de l'exécution : 0.0099
Temps d'exécution : 27249.643 s
À la fin de l'algorithme :
- max_ecarts : 0.104
- somme_ecarts : 7.904
Comparaison avec AS :
- max_ecarts : 0.134
- somme_ecarts : 30.925
```

On constate clairement que les écarts entre les fonctions de valeur successives de MQL a fortement diminué par rapport au premier apprentissage, de même que les écarts entre les fonctions de valeur de MQL et de AS. On voit donc qu'avec 20 fois plus d'itérations on a davantage convergé vers l'équilibre de Nash : on peut supposer que MQL tel que nous l'avons implémenté peut effectivement converger vers l'équilibre de Nash avec un temps d'exécution très long. Au vu de l'amélioration constatée entre les deux apprentissages, nous estimons qu'environ 100 millions itérations seraient nécessaires pour converger autant que AS, c'est-à-dire à 10^{-12} près, ce qui prendrait environ 40 heures de temps d'exécution.

Nous avons par ailleurs déterminé que les stratégies de MQL et de AS différaient sur 276 des 760 états. Lors du premier apprentissage, ils différaient sur 384 états.

```
Minimax-Q-learning 2
Stratégie de l'adversaire : stratégie obtenue par AS
Paramètres : nombre d'itérations = 20 * 10 ** 6, explor = 0.3,
decay = 0.01 ** (1. / ( 2 *10**4))
Taux d'apprentissage à la fin de l'exécution : 0.0099
Temps d'exécution : 27244.263 s
À la fin de l'algorithme :
- max_ecarts : 1.908
- somme_ecarts : 254.197
Comparaison avec AS :
- max_ecarts : 1.899
- somme_ecarts : 341.006
```

Après apprentissage de MQL contre AS, on peut voir que même en 20 millions d'itérations les écarts entre les fonctions de valeur successives de MQL et les écarts entre MQL et AS restent très importants en raison des états jamais modifiés, auxquels l'algorithme n'accède jamais car la stratégie de AS les évite (notamment les coins). En revanche pour les états que l'algorithme a modifié, la valeur de jeu et les probabilités des actions de MQL sont très proches de celles obtenues par AS.

5.2.4 Seconde phase de comparaisons

```
Test sur 1 000 000 de parties : MQL2 versus AS
Récompenses totales : {0: 0.338, 1: -0.338}
Taux de victoire : {0: 0.506094660482816, 1: 0.49390533951718396}
Nombre de buts : 67272
```

On voit à nouveau que même sans avoir convergé vers l'équilibre de Nash, la stratégie obtenue par MQL contre AS en 20 millions d'itérations est optimale face à la stratégie obtenue par AS : la stratégie de MQL gagne même légèrement plus souvent (50.6% de victoires).

```
Test sur 1 000 000 de parties : MQL1 versus AS
Récompenses totales : {0: 0.178, 1: -0.178}
Taux de victoire : {0: 0.5003940227492379, 1: 0.499605977250762}
Nombre de buts : 67255
```

La stratégie obtenue par MQL face à la stratégie aléatoire en 20 millions d'itérations est également optimale contre la stratégie de AS, avec des taux de victoires quasiment identiques. C'est d'autant plus remarquable qu'encore une fois MQL n'avait pas parfaitement convergé.

```
Test sur 1 000 000 de parties : MQL1 versus aléatoire
Récompenses totales : {0: 1.0398, 1: -1.0398}
Taux de victoire : {0: 0.9877023503971222, 1: 0.012297649602877872}
Nombre de buts : 98962
```

```
Test sur 1 000 000 de parties : MQL2 versus aléatoire
Récompenses totales : {0: 0.721, 1: -0.721}
Taux de victoire : {0: 0.9353993494759667, 1: 0.06460065052403324}
Nombre de buts : 66408
```

Enfin, sans surprise, les deux stratégies obtenues par MQL sont très efficaces contre la stratégie aléatoire, même celle obtenue contre AS.

6 Conclusion

Dans ce projet, nous avons conçu et implémenté une modélisation générique des jeux stochastiques en tenant compte des contraintes techniques et des besoins des utilisateurs, et nous avons implémenté des algorithmes permettant de résoudre une sous-classe des jeux stochastiques, ceux à somme nulle à deux joueurs. Nous avons ensuite implémenté avec cette modélisation les jeux que nous avions présentés, et testé avec succès ces algorithmes sur eux.

Concernant le problème de la convergence de minimax-Q-learning face à la stratégie obtenue par Shapley, nous émettons l'hypothèse suivante : en intégrant dans l'algorithme une probabilité d'exploration pour la stratégie de l'adversaire (exactement comme pour le joueur apprenant), il devrait être possible de réduire le nombre d'états inexplorés et donc de converger davantage vers la fonction de valeur optimale.

A *StochasticGames*

```
import itertools as it

class StochasticGame:
    """
    Generic class for stochastic games
    """

    def states(self):
        """Return the list of all states.

        :rtype: list of (tuples of) integers/strings
        """
        raise(NotImplementedError)


    def players(self):
        """Return the list of all players.

        :rtype: list of integers/strings
        """
        raise(NotImplementedError)


    def actions(self, player, state):
        """Return the list of given player's possible actions in given state.

        :param player: player ID
        :param state: state
        :rtype: list of (tuples of) integers/strings
```

```
    """
    raise(NotImplementedError)


def nb_players(self):
    """Return the number of players.

    :rtype: integer
    """
    return len(self.players())


def gamma(self):
    """Return the discount factor.

    :rtype: float in [0, 1[
    """
    raise(NotImplementedError)


def transition(self, state, actions):
    """Return a probability distribution for the next state.

    :param state: current state
    :param actions: dictionary (key: player; value: action)
    :rtype: dictionary {state: probability}
    """
    raise(NotImplementedError)


def _reward(self, player, state, actions):
    """Return the immediate reward of a player given a state and the actions of all players.
    You should implement either this function or rewards(self, state, actions).

    :param player: player ID
    :param state: current state
    :param actions: dictionary (key: player; value: action)
    :rtype: integer or float
    """
    raise(NotImplementedError)


def rewards(self, state, actions):
    """Return the immediate reward of all players given a state and the actions of all players.
    You should implement either this function or _reward(self, player, state, actions).

    :param state: current state
    :param actions:  dictionary (key: player; value: action)
    :rtype:  dictionary (key: player; value: reward)
    """
    return {player: self._reward(player, state, actions) for player in self.players()}
```

```
    def initial_state(self):
        """Return a probability distribution for the initial state.

        :rtype: dictionary {state: probability}
        """
        raise(NotImplementedError)


    def check_transitions(self, epsilon):
        """
        verify that all transitions are valid probability distributions.

        :param epsilon: error rate
        """
        for state in self.states():
            for actions in it.product(self.actions(player, state) for player in self.players()):
                if abs(1 - sum(self.transition(state, actions).values())) > epsilon:
                    return False
        return True
```

B *NullSum2PlayerStochasticGame*

```
from StochasticGame import StochasticGame

class NullSum2PlayerStochasticGame(StochasticGame):
    """
    Specific class for null sum 2-player stochastic games.
Player 0 must maximize his rewards, player 1 must minimize them.
    """

    def players(self):
        return [0, 1]


    def player0_reward(self, state, actions):
        raise(NotImplementedError)


    def rewards(self, state, actions):
        r = self.player0_reward(state, actions)
        return {0:r, 1:-r}
```

C Jeu de football simplifié

```
from NullSum2PlayerStochasticGame import NullSum2PlayerStochasticGame

class Soccer(NullSum2PlayerStochasticGame):
```

```
movement = {"N":(-1, 0), "S":(1, 0), "E":(0, 1), "W":(0, -1), "stand":(0, 0)}
list_actions = list(movement.keys())


def __init__(self):
    self.cells = [(i, j) for i in range(1, 5) for j in range(1, 6)]
    self.list_states = [(c1, c2, b) for c1 in self.cells for c2 in self.cells \
        for b in [1, 2] if c1 != c2]
    self.starting_positions = ((3, 2), (2, 4))
    self.left_goal_positions = ((2, 0), (3, 0))
    self.right_goal_positions = ((2, 6), (3, 6))
    self.goal_positions = self.left_goal_positions + self.right_goal_positions


def next_position(self, position, action):
    return tuple(map(sum, zip(position, self.movement[action])))


def states(self):
    return self.list_states


def actions(self, player, state):
    def allowed(action):
        position = self.next_position(state[player], action)
        return position in self.cells or \
           (position in list(self.goal_positions) and state[2] == player+1)
    return [action for action in self.list_actions if allowed(action)]


def gamma(self):
    return .9


def initial_state(self):
    return {(*self.starting_positions, 1): .5, (*self.starting_positions, 2): .5}


def transition(self, state, actions):

    # desired positions
    pos1, pos2, b = state
    dest1 = self.next_position(pos1, actions[0])
    dest2 = self.next_position(pos2, actions[1])

    # goal
    if (b == 1 and dest1 in self.goal_positions) or \
     (b == 2 and dest2 in self.goal_positions):
        return self.initial_state()
```

```
        # frontal shock: players try to go to each other's position
        if dest1 == pos2 and dest2 == pos1:
            return {(pos1, pos2, 1): .5, (pos1, pos2, 2): .5}

        # blocking: players try to go to the same position
        if dest1 == dest2:

            # obstruction: one of the players chooses not to move
            if dest1 == pos1:
                return {(pos1, pos2, 1): 1}
            if dest2 == pos2:
                return {(pos1, pos2, 2): 1}

            # players rush to another position
            return {(dest1, pos2, 1): .5, (pos1, dest2, 2): .5}

        # one of the players "follows" the other
        if dest1 == pos2: # and dest2 not in [pos1, pos2]
            return {(dest1, dest2, b): .5, (pos1, dest2, 2): .5}
        if dest2 == pos1: # and dest1 not in [pos1, pos2]
            return {(dest1, dest2, b): .5, (dest1, pos2, 1): .5}

        # normal movement
        return {(dest1, dest2, b): 1}


    def player0_reward(self, state, actions):

        # desired positions
        pos1, pos2, b = state
        dest1 = self.next_position(pos1, actions[0])
        dest2 = self.next_position(pos2, actions[1])

        # goal
        if (b == 1 and dest1 in self.right_goal_positions) or \
         (b == 2 and dest2 in self.right_goal_positions):
            return 1

        if (b == 2 and dest2 in self.left_goal_positions) or \
         (b == 1 and dest1 in self.left_goal_positions):
            return -1

        return 0
```

Références

[1] F. Garcia, "Processus Décisionnels de Markov," in *Processus Décisionnels de Markov en Intelligence Artificielle*, O. Sigaud and O. Buffet, Eds. Hermes, 2008, ch. 1, pp. 17–50.

[2] R. Bellman, *Dynamic Programming*, 1st ed. Princeton, NJ, USA : Princeton University Press, 1957.

[3] A. Burkov and B. Chaib-Draa, "Une introduction aux jeux stochastiques," in *Processus Décisionnels de Markov en Intelligence Artificielle*, O. Sigaud and O. Buffet, Eds. Hermes, 2008, ch. 4, pp. 135–176.

[4] J. Filar and K. Vrieze, *Competitive Markov Decision Processes*. Springer-Verlag Berlin, Heidelberg, 1997.

[5] L. Shapley, "Stochastic games," *Proceedings of the National Academy of Sciences of the U. S. A.*, vol. 39, p. 1095–1100, 1953.

[6] M. L. Littman, "Markov games as a framework for multi-agent reinforcement learning," *Proceedings of the 11th International Conference on Machine Learning (ML-94)*, p. 157–163, 1994.

[7] C. Y. T. Ma, D. K. Y. Yau, X. Lou, and N. S. V. Rao, "Markov Game Analysis for Attack-Defense of Power Networks," *IEEE Transactions on Power Systems*, vol. 28, pp. 1676 – 1686, 2013.

[8] F. Garcia and O. Sigaud, "Apprentissage par renforcement," in *Processus Décisionnels de Markov en Intelligence Artificielle*, O. Sigaud and O. Buffet, Eds. Hermes, 2008, ch. 2, pp. 51–84.

[9] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, pp. 279–292, 1992.

[10] O. Gies and B. Chaib-Draa, "Apprentissage de la coordination multiagent : une méthode basée sur le Q-learning par jeu adaptatif," *Revue d'Intelligence Artificielle*, pp. 20(2–3) :385–412, 2006.

[11] C. Szepesvári and M. L. Littman, "Generalized Markov decision processes : Dynamic-programming and reinforcement-learning algorithms," *Proceedings of International Conference of Machine Learning '96, Bari*, 1996.