

Devoir 3 : Puissance 4 (*Monte Carlo*)

Informations pratiques

Ce devoir est individuel. Il vous est demandé de résoudre ce problème individuellement. Vous êtes autorisé.e, et même encouragé.e, à échanger des idées avec d'autres étudiant.e.s sur la façon d'aborder ce devoir. En revanche, vous devez rédiger votre solution individuellement : ne partagez pas votre production. Nous serons intransigeant.e.s si nous observons des similitudes dans votre code ou votre rapport, lesquels passeront dans les logiciels anti-plagiat de Gradescope.

Echéance : Mercredi 27 novembre 2024 à 18h00.

Informations pratiques

- Soumission sur <https://www.gradescope.com/>, connectez-vous avec vos identifiants UCLouvain.
- Si vous êtes inscrits sur la page Moodle du cours, vous devriez voir apparaître LEPL1108 dans la liste de vos cours. Si ce n'est pas le cas, vous devriez pouvoir rejoindre le cours en vous servant du code Z3BEBY.
- Ce devoir est divisé en quatre sections. Les sections 1, 2 et 4 doivent être complétées en répondant aux questions par écrit et en soumettant l'énoncé complété en PDF sur Gradescope dans la partie Devoir 3: Puissance 4 PDF. Le code de la section 3 doit être soumis sur Gradescope dans la partie Devoir 3: Puissance 4 AI.
- Un fichier Latex intitulé `Devoir3.tex` est disponible sur Moodle pour faciliter le remplissage de l'énoncé du devoir. Ne modifiez pas la taille et/ou la position des cadres de réponse pour éviter des problèmes de lecture par Gradescope.
- Pour les codes, respectez le modèle fourni, vous risquez sinon d'avoir des problèmes de codage ou des problèmes de lecture du code par Gradescope.
- Sur Moodle, vous trouverez un fichier "squelette" `montecarlo.py` à compléter et un fichier `connect4.py` qui seront fournis pour les sections 2 et 4 ainsi qu'un fichier "squelette" `ai_student.py` à compléter pour la section 3.
- Pour la section 3, les seuls packages autorisés sont : `numpy`, `math` et `random`.
- Pour ce devoir l'appellation joueur 1/player 1 désigne le joueur qui entame la partie.

Introduction

Nous vous proposons de jouer un jeu que vous connaissez sans doute : Puissance 4. Pour celle et ceux qui ne sont pas familier.e.s avec ce jeu, les règles peuvent être trouvées sur ce site). Etant toutes des grands amateur.ice.s de calcul probabiliste, nous utiliserons la méthode dite de Monte Carlo pour calculer le pourcentage de victoire attendu à Puissance 4 pour différentes stratégies de jeu.

1 Borne inférieure

Afin de calculer exactement des probabilités de succès ou d'échec dans ce jeu, il faudrait explorer toutes les parties possibles. Or, on soupçonne que ce nombre est important. Dans cette section, on calcule une borne inférieure sur le nombre de parties possibles, afin de soutenir notre intuition et justifier le recours à la méthode de Monte Carlo.

On propose un calcul simple de borne inférieure. Peu importe que cette borne soit proche ou non du nombre exact de parties possibles (que vous pouvez trouver en quelques clics sur Google). Notre borne peut être grossière, il suffit qu'elle soit assez grande pour justifier notre recours à Monte Carlo.

On considère l'état du plateau représenté en Figure 1, et on demande de dénombrer toutes les parties possibles qui ont pu mener à un tel état du plateau de jeu. Bien que de nombreuses bornes inférieures puissent être proposées, ici on s'attend au nombre exact de parties ayant mené au plateau de la Figure 1 – il n'y a qu'une seule bonne réponse. Justifiez votre réponse en expliquant votre raisonnement.

NB : Afin que les choses soient claires, deux parties sont considérées comme différentes si les coups ne sont pas joués dans le même ordre.

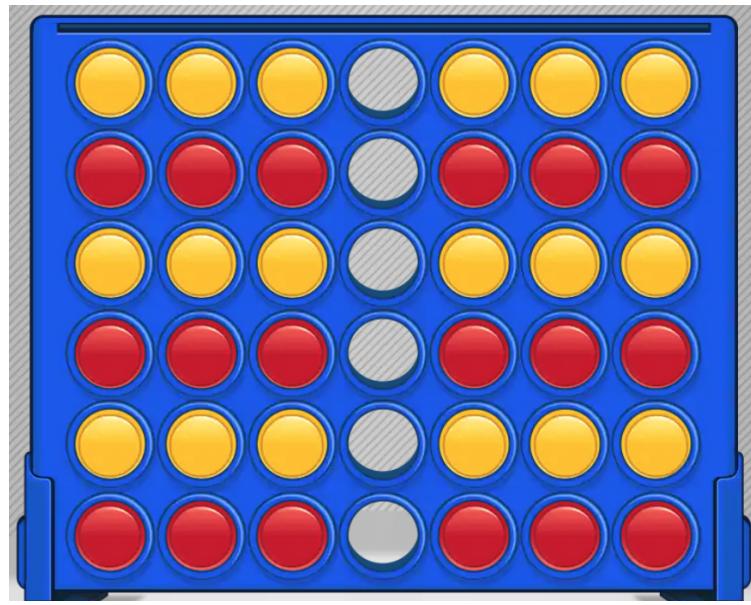


FIGURE 1 – Plateau de jeu considéré pour le calcul de borne inférieure

Solution: Insérez votre réponse ci-dessous.

2 Méthode de Monte Carlo

Dans cette section, on considère une stratégie de jeu très basique : `ai_random`. Cette stratégie fonctionne de la manière suivante :

- S'il y a un coup gagnant à jouer (c'est-à-dire une ligne de trois jetons de sa couleur, avec une place libre à côté), elle le joue. On parle d'`attack move`.
- Sinon, s'il y a un coup gagnant pour l'adversaire, elle le joue afin de bloquer une potentielle victoire de l'adversaire au prochain tour. On parle de `defense move`.
- Sinon, elle joue au hasard parmi les colonnes non-remplies.

On s'intéresse à la question suivante : si la stratégie `ai_random` joue contre elle-même, quel est l'avantage à entamer la partie ? Plus précisément, si les deux joueurs suivent la stratégie `ai_random`, quelle est la probabilité que le joueur 1 (celui qui commence) gagne, la probabilité que le joueur 2 gagne, et la probabilité d'`ex-æquo` ?

La borne inférieure obtenue en section 1 montre que calculer ces probabilités de manière exacte n'est pas gérable en temps de calcul sur votre ordinateur. C'est pourquoi on a recours à la méthode dite de Monte Carlo afin d'approximer ces probabilités.

La méthode de Monte Carlo consiste à estimer ces probabilités en simulant un grand nombre de parties, et en comptant le nombre de victoires et d'`ex-æquo` pour ces parties simulées. On s'intéresse aussi à l'évolution de l'estimé obtenu en fonction du nombre de parties jouées : plus le nombre de parties jouées est élevé, plus l'estimé sera bon.

On vous demande de calculer le pourcentage de victoires du joueur 1 et le pourcentage d'`ex-æquo` pour 10, 100, 1000 et 10000 parties. A chaque fois, répétez l'expérience dix fois. Représentez sur un graphique le résultat obtenu pour chacune des dix expériences, ainsi que la moyenne de ces dix expériences. Pour ce faire, on vous fournit :

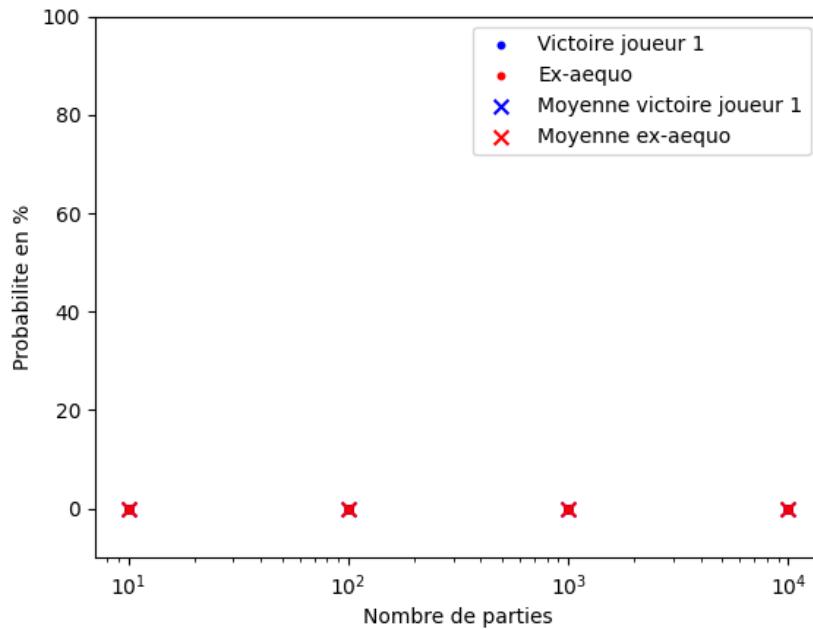
- `connect4.py` : fichier qui réunit toutes les fonctions nécessaires afin de simuler une partie de Puissance 4 :
 - `ai_random` : la stratégie basique considérée dans cette section.
 - `update_board` : met à jour le tableau de jeu à l'aide du coup reçu en argument.
 - `print_board` : permet une visualisation de l'état du jeu. Utile pour les sections 3 et 4 lorsque vous coderez votre propre AI, et pour débugger.
 - `check_win` : vérifie si le dernier coup joué est gagnant.
 - `run_game` : joue une partie de Puissance 4, et retourne 1 si le 1e joueur l'emporte, 2 si c'est le second et 0 si c'est ex-æquo.

Pour cette section, vous ne devez pas modifier ce fichier, mais seulement utiliser la fonction `run_game` afin de simuler des parties.

- `montecarlo.py` : script à compléter qui simule des parties puis représente graphiquement les résultats obtenus. Le format du graphique est fixé, on vous demande de ne pas le modifier afin de faciliter la correction. La partie du code à compléter vise à calculer les tableaux `win1` et `draw` : les résultats des simulations.

Insérez votre graphique dans le cadre ci-dessous. Il est normal que votre code prenne ~ 15 min à tourner sur votre machine pour 10^4 parties.

Solution: Insérez votre graphique ci-dessous, à la place de ce graphique vide.



Sur base du graphique obtenu, répondez aux questions suivantes :

- Y a-t-il un avantage à commencer la partie ? Justifiez votre réponse sur base de votre graphique, en mentionnant les probabilités moyennes obtenues pour 10^4 parties.

Solution: Insérez votre réponse ci-dessous.

- Soit n le nombre de parties simulées. Pour un n fixé, on s'intéresse à la distance typique des résultats des dix expériences par rapport à leur moyenne. Sur base de votre graphique, vous observez que cette distance typique évolue (à une constante multiplicative près) en :

$$(i) n^2 \quad (ii) n \quad (iii) \sqrt{n} \quad (iv) 1/\sqrt{n} \quad (v) 1/n^2$$

Justifiez votre réponse sur base de votre graphique. Cette notion sera approfondie en cours.

Solution: Insérez votre réponse ci-dessous.

3 Codez votre intelligence artificielle

Maintenant que nous avons analysé ce qui se passerait avec deux joueurs qui jouent une stratégie aléatoire `ai_random`, nous vous proposons d'aller un cran plus loin et de coder votre propre intelligence artificielle (IA) qui joue à Puissance 4. Pour tester le bon fonctionnement de votre IA, nous avons préparé une série de tests sur Gradescope de situations critiques pendant une partie de Puissance 4 où votre IA devrait prendre la meilleure décision possible (c.-à-d. choisir de jouer la colonne qui maximise la chance du joueur à gagner). Votre but est de jouer le meilleur coup possible en prenant en compte l'état actuel du plateau de jeu avant votre tour.

Il vous est demandé d'implémenter en Python votre IA `ai_student.py`, sous la forme d'une fonction dont la signature est donnée ci-dessous :

```
1 def ai_student(board, player):
2
3     # to do
4
5     return colonne_choisie
```

Inputs

- `board` est un tableau (array) numpy de taille 6×7 qui ne peut contenir que les chiffres 0, 1 ou 2 (0 signifiant que la case est vide, 1 signifiant que la case est remplie par un jeton du joueur 1 et 2 signifiant que la case est remplie par un jeton du joueur 2. Les indexes du tableau font de haut en bas et de gauche à droite, c'est-à-dire que $(0, 0)$ correspond au coin en haut à gauche et $(6, 7)$ correspond au coin en bas à droite. Le tableau représente l'état actuel du plateau de jeu à votre tour. Les 0, 1 et 2 sont encodés comme des entiers (non pas des strings) dans le tableau.

Exemple : Le plateau de jeu de la Figure 2 est représenté par le tableau suivant :

$$\left[\begin{bmatrix} 0, & 0, & 0, & 0, & 0, & 0, & 0 \end{bmatrix}, \right. \\ \begin{bmatrix} 0, & 0, & 0, & 0, & 0, & 0, & 0 \end{bmatrix}, \\ \begin{bmatrix} 0, & 0, & 0, & 0, & 0, & 0, & 0 \end{bmatrix}, \\ \begin{bmatrix} 0, & 0, & 2, & 0, & 0, & 0, & 0 \end{bmatrix}, \\ \begin{bmatrix} 0, & 0, & 2, & 1, & 0, & 0, & 0 \end{bmatrix}, \\ \left. \begin{bmatrix} 0, & 1, & 2, & 1, & 0, & 0, & 0 \end{bmatrix} \right]$$

- `player` est un entier qui peut prendre la valeur 1 ou 2 et qui représente si nous jouons pour le joueur 1 ou le joueur 2. *Exemple* : 1

Output

- `colonne_choisie` est un entier (compris entre 0 et 6) qui représente la colonne que notre IA à chosie de jouer. *Exemple* : 2 (la troisième colonne)

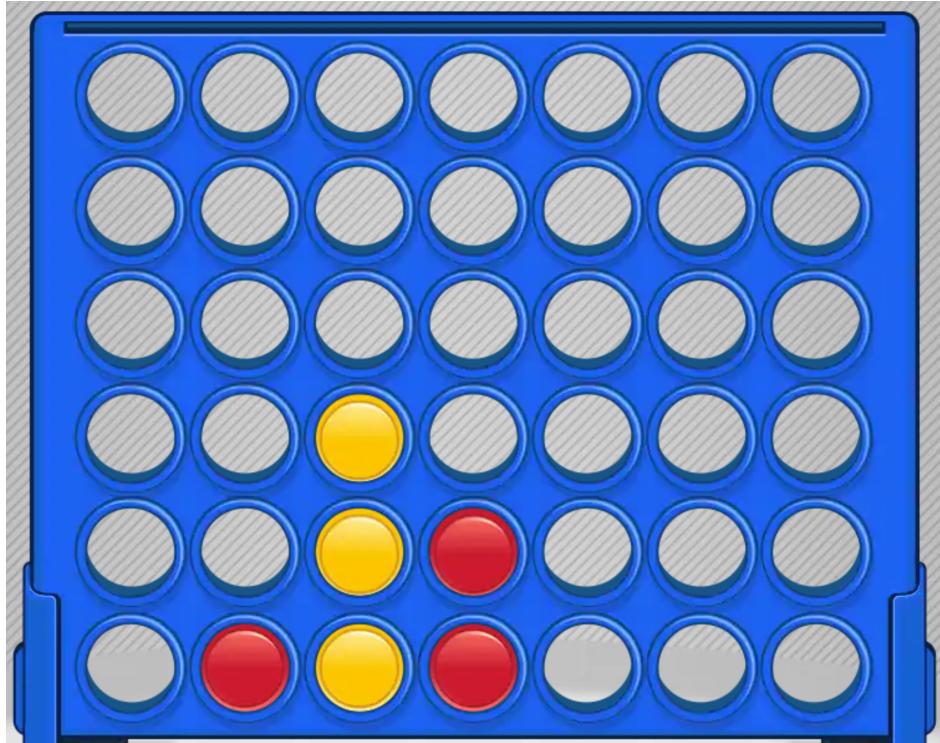


FIGURE 2 – Illustration de l'exemple utilisé pour expliquer le code `ai_student`. Le joueur 1 est représenté par la couleur rouge et le joueur 2 est représenté par la couleur jaune. Le joueur 1 doit jouer la colonne 3 pour éviter de perdre la partie.

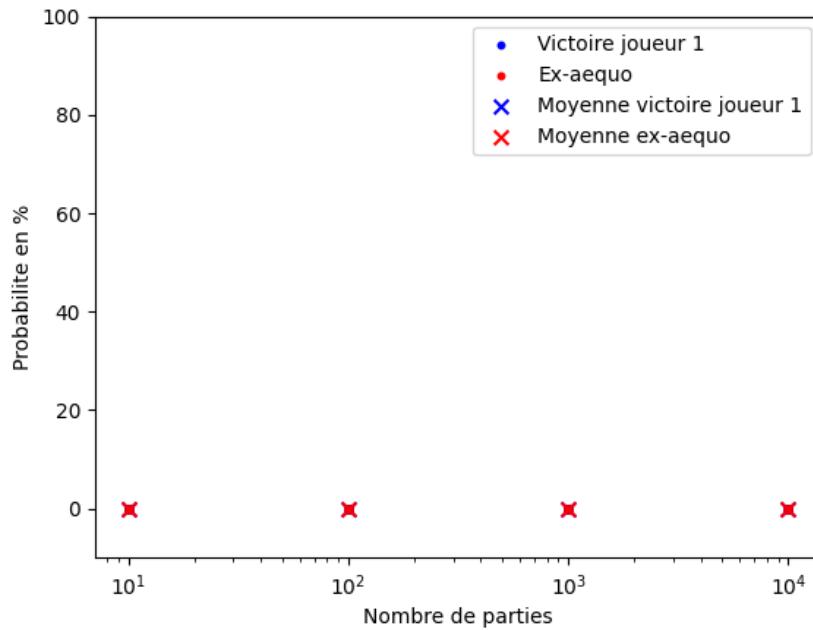
4 Monte Carlo avec votre intelligence artificielle

Si vous arrivez à cette partie du devoir, c'est que votre IA a (partiellement) réussi les tests sur Gradescope. Bravo ! Pour observer l'amélioration que votre IA apporte par rapport à l'IA aléatoire évaluée dans la section 2, nous vous proposons de les faire jouer l'une contre l'autre. Pour ce faire, nous aurons à nouveau recours à l'algorithme de Monte Carlo. Dans cette section, vous avez l'occasion de nous convaincre de la qualité de votre IA de manière objective en générant le même graphique qu'en section 2.

Maintenant que vous avez codé votre IA `ai_student`, il vous suffit de modifier la fonction `run_game` dans le fichier `connect4.py`. Par exemple, pour que le joueur 1 suive votre stratégie, remplacez la ligne `move1 = ai_random(the_board, 1)` par `move1 = ai_student(the_board, 1)`. Il vous est alors demandé de générer le même graphique qu'en section 2 pour les deux cas suivants :

Cas 1 : Le joueur 1 joue ai_student. et le joueur 2 joue ai_random.

Solution: Insérez le graphe et mentionnez les probabilités moyennes obtenues pour 10^4 parties.



Cas 1 : Le joueur 1 joue ai_random. et le joueur 2 joue ai_student.

Solution: Insérez le graphe et mentionnez les probabilités moyennes obtenues pour 10^4 parties.

