

### Devoir 3 : Puissance 4 (*Monte Carlo*)

---

## Informations pratiques

*Ce devoir est individuel.* Nous vous encourageons à échanger vos idées entre vous sur la façon d'aborder ce devoir. En revanche, votre solution doit être rédigée individuellement : ne partagez pas votre production ! Nous serons intransigeants si nous observons des similitudes dans votre code ou votre rapport, lesquels passeront dans les logiciels anti-plagiat de Gradescope.

*Echéance :* Mercredi 29 Décembre 2023 à 18h00.

## Soumission

- Soumission sur <https://www.gradescope.com/>, connectez-vous avec vos identifiants UCLouvain.
- 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 6GGV63.
- 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 modifier 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 ou la joueuse qui entame la partie.
- **Pondération des points : Les sections 3 et 4 du devoir sont des questions de dé-passement et compteront comme bonus pour ce devoir. Un bonus supplémentaire sera également attribué aux étudiants et étudiantes qui obtiendront les meilleurs scores pour les sections 3 et 4.**

# Introduction

Nous vous proposons de jouer un jeu que vous connaissez toutes et tous : le Puissance 4. Pour celles et ceux qui ne sont pas familiers avec ce jeu, les règles peuvent être trouvées sur le site suivant : <https://www.regles.com/jeux/puissance-4.html>. Etant toutes et tous des grands amateurs 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.

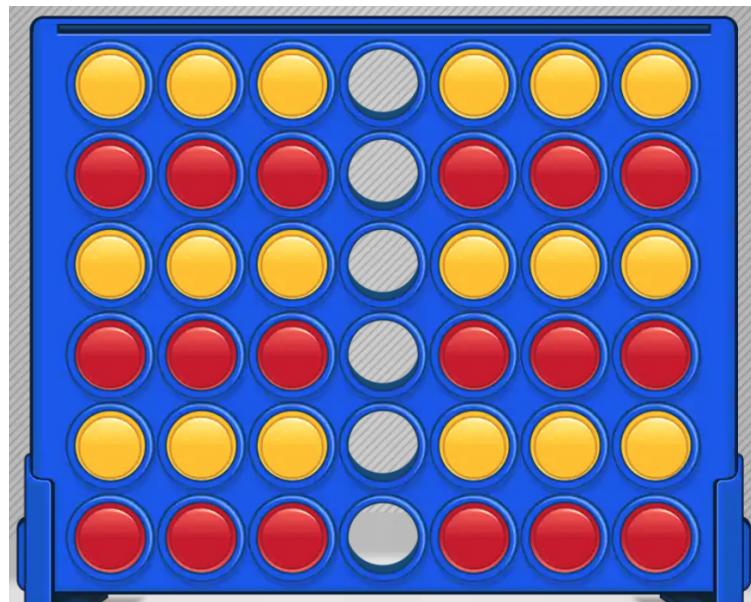


FIGURE 1 – Plateau de jeu considéré pour le calcul de borne inférieure

**Solution:** Le joueur jaune n'a aucune influence sur le jeu en raison de la disposition du plateau, il est obligé de suivre le joueur rouge.

Nous nous intéressons donc uniquement au joueur rouge. On constate que le joueur rouge joue 18 coups au cours de sa partie.

Il y a donc 18 permutations possibles pour les coups joués par le joueur rouge. Cependant, on constate que certaines permutations défient la gravité.

En effet, on ne peut pas permute l'ordre de coups situés l'un au-dessus de l'autre. Il y a donc 3 coups par colonne, et donc  $3!$  permutations à supprimer par colonne.

Pour éliminer les permutations impossibles, il nous suffit de diviser le nombre total de permutations par le nombre de permutations indésirables. On obtient donc  $\frac{18!}{(3!)^6} = 1.2 \times 10^{11}$  parties possibles. L'exposant 6 est dû au fait qu'il y a 6 colonnes.

## 2 Méthode de Monte Carlo

Dans cette section, nous considérons 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), il est joué, c'est un **attack move**.
- Sinon, s'il y a un coup gagnant pour l'adversaire, ce coup là est joué, afin de bloquer une potentielle victoire de l'adversaire au prochain tour, c'est un **defense move**.
- Sinon, un coup est fait au hasard parmi les colonnes non-remplies.

Nous nous intéressons 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 nous avons 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. Nous nous intéressons 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.

Il vous est demandé 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, les fichiers suivants vous sont fournis :

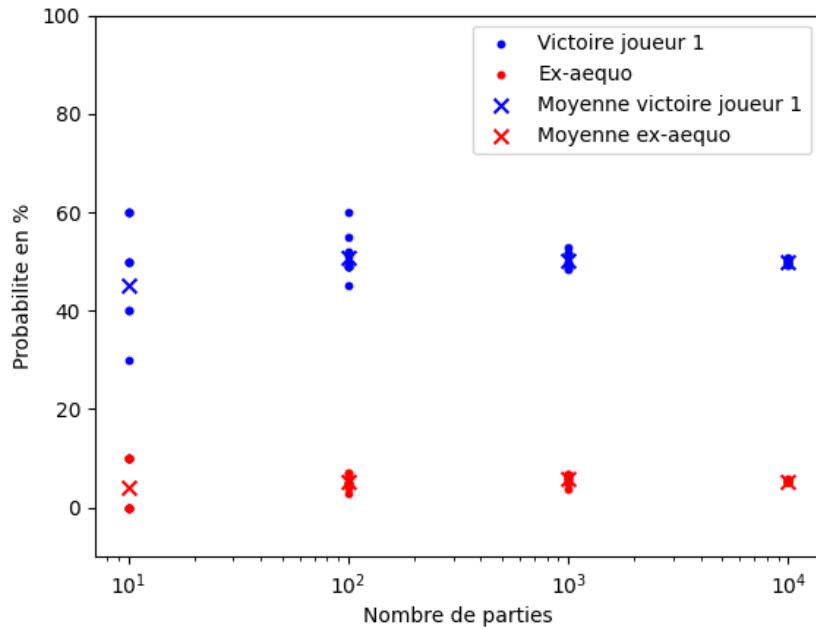
- `connect4.py` : fichier qui réunit toutes les fonctions nécessaires afin de simuler une partie de Puissance 4 :
  - `ai_random` : la stratégie basique considérée dans cette section.
  - `update_board` : met à jour le tableau de jeu à l'aide du coup reçu en argument.
  - `print_board` : permet une visualisation 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  $\sim 15$  min à tourner sur votre machine pour  $10^4$  parties.

**Solution:** Voici le graphique pour *ai\_random* jouant contre *ai\_random*



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:** Oui il y a un grand avantage à commencer la partie en 1e comme on peut le voir sur le graphe le joueur qui commence en premier a bien plus de chance de gagner que le joueur 2.

- 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:** On peut voir que la distance évolue en  $1/\sqrt{n}$

### 3 Codez votre propre 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 import random
2
3 ROW_COUNT = 6
4 COLUMN_COUNT = 7
5
6 database = {}
7
8 def is_valid_move(board, col):
9     if board[0][col] != 0:
10         return False
11     return True
12
13 def get_row(board, col):
14     for r in range(ROW_COUNT):
15         row = ROW_COUNT - r - 1
16         if board[row][col] == 0:
17             return row
18
19 def is_winning_move(board, player, move):
20     row = get_row(board, move)
21     board[row][move] = player
22     if check_win(board, player):
23         board[row][move] = 0
24         return True
25     board[row][move] = 0
26     return False
27
28 def check_win(board, player):
29     # Check horizontal
30     for col in range(COLUMN_COUNT):
31         for row in range(ROW_COUNT):
32             if col+3 < 7:
33                 if board[row][col] == player and board[row][col+1] == player and
34                 board[row][col+2] == player and board[row][col+3] == player:
35                     return True
36             # Check vertical
37             if row+3 < 6:
38                 if board[row][col] == player and board[row+1][col] == player and
39                 board[row+2][col] == player and board[row+3][col] == player:
40                     return True
41             # Check down right diagonals
42             if col+3 < 7 and row+3 < 6:
43                 if board[row][col] == player and board[row+1][col+1] == player and
```

```

        board[row+2][col+2] == player and board[row+3][col+3] == player:
            return True
    # Check down left diagonals
    if col-3 >= 0 and row+3 < 6:
        if board[row][col] == player and board[row+1][col-1] == player and
board[row+2][col-2] == player and board[row+3][col-3] == player:
            return True
47
48 def evaluate_board(board, player):
49     score = 0
50     opponent = 1 if player == 2 else 2
51     for i in range(four_in_a_row(board, player)):
52         score += 1000
53     for i in range(four_in_a_row(board, opponent)):
54         score -= 1000
55     for i in range(three_in_a_row(board, player)):
56         score += 100
57     for i in range(three_in_a_row(board, opponent)):
58         score -= 100
59     for i in range(two_in_a_row(board, player)):
60         score += 10
61     for i in range(two_in_a_row(board, opponent)):
62         score -= 10
63     for i in range(one_in_a_row(board, player)):
64         score += 1
65     for i in range(one_in_a_row(board, opponent)):
66         score -= 1
67     return score
68
69 def four_in_a_row(board, player):
70     # Check horizontal
71     score = 0
72     for col in range(COLUMN_COUNT):
73         for row in range(ROW_COUNT):
74             if col+3 < 7:
75                 if board[row][col] == player and board[row][col+1] == player and
board[row][col+2] == player and board[row][col+3] == player:
76                     score += 1
77                 # Check vertical
78                 if row+3 < 6:
79                     if board[row][col] == player and board[row+1][col] == player and
board[row+2][col] == player and board[row+3][col] == player:
80                         score += 1
81                 # Check down right diagonals
82                 if col+3 < 7 and row+3 < 6:
83                     if board[row][col] == player and board[row+1][col+1] == player and
board[row+2][col+2] == player and board[row+3][col+3] == player:
84                         score += 1
85                 # Check down left diagonals
86                 if col-3 >= 0 and row+3 < 6:
87                     if board[row][col] == player and board[row+1][col-1] == player and
board[row+2][col-2] == player and board[row+3][col-3] == player:
88                         score += 1
89     return score
90
91 def three_in_a_row(board, player):
92     score = 0

```

```

93     for col in range(7):
94         for row in range(6):
95             # checking horizontal
96             if col+3 < 7:
97                 if board[row][col] == player and board[row][col+1] == player and
board[row][col+2] == player and board[row][col+3] == 0:
98                     score += 1
99             # checking vertical
100            if row+3 < 6:
101                if board[row][col] == player and board[row+1][col] == player and
board[row+2][col] == player and board[row+3][col] == 0:
102                    score += 1
103            # checking down left diag
104            if col-3 >= 0 and row+3 < 6:
105                if board[row][col] == player and board[row+1][col-1] == player and
board[row+2][col-2] == player and board[row+3][col-3] == 0:
106                    score += 1
107            # checking down right diag
108            if col+3 < 7 and row+3 < 6:
109                if board[row][col] == player and board[row+1][col+1] == player and
board[row+2][col+2] == player and board[row+3][col+3] == 0:
110                    score += 1
111    return score
112
113 def two_in_a_row(board, player):
114     score = 0
115     for col in range(7):
116         for row in range(6):
117             # checking horizontal
118             if col+3 < 7:
119                 if board[row][col] == player and board[row][col+1] == player and
board[row][col+2] == 0 and board[row][col+3] == 0:
120                     score += 1
121             # checking vertical
122             if row+3 < 6:
123                 if board[row][col] == player and board[row+1][col] == player and
board[row+2][col] == 0 and board[row+3][col] == 0:
124                     score += 1
125             # checking down left diag
126             if col-3 >= 0 and row+3 < 6:
127                 if board[row][col] == player and board[row+1][col-1] == player and
board[row+2][col-2] == 0 and board[row+3][col-3] == 0:
128                     score += 1
129             # checking down right diag
130             if col+3 < 7 and row+3 < 6:
131                 if board[row][col] == player and board[row+1][col+1] == player and
board[row+2][col+2] == 0 and board[row+3][col+3] == 0:
132                     score += 1
133     return score
134
135 def one_in_a_row(board, player):
136     score = 0
137     for col in range(7):
138         for row in range(6):
139             # checking horizontal
140             if col+3 < 7:
141                 if board[row][col] == player and board[row][col+1] == 0 and board[
```

```

142     row][col+2] == 0 and board[row][col+3] == 0:
143         score += 1
144     # checking vertical
145     if row+3 < 6:
146         if board[row][col] == player and board[row+1][col] == 0 and board[
147             row+2][col] == 0 and board[row+3][col] == 0:
148             score += 1
149     # checking down left diag
150     if col-3 >= 0 and row+3 < 6:
151         if board[row][col] == player and board[row+1][col-1] == 0 and
152             board[row+2][col-2] == 0 and board[row+3][col-3] == 0:
153                 score += 1
154     # checking down right diag
155     if col+3 < 7 and row+3 < 6:
156         if board[row][col] == player and board[row+1][col+1] == 0 and
157             board[row+2][col+2] == 0 and board[row+3][col+3] == 0:
158                 score += 1
159
160     return score
161
162 def is_terminal_state(board):
163     # checking if a player won
164     if four_in_a_row(board, 1) or four_in_a_row(board, 2):
165         return True
166     # checking if board is full
167     for col in range(7):
168         if board[0][col] == 0:
169             return False
170     return True
171
172 score_matrix = [0.2, 0.5, 1.0, 3, 1.0, 0.5, 0.2]
173
174 def minimax(board, player, depth, alpha, beta, maximizingPlayer):
175     opponent = 1 if player == 2 else 2
176     if str(board) in database:
177         return database[str(board)]
178     if depth == 0 or is_terminal_state(board):
179         return evaluate_board(board, player)
180
181     if maximizingPlayer:
182         value = -float('inf')
183         for col in range(7):
184             if is_valid_move(board, col):
185                 row = get_row(board, col)
186                 board[row][col] = player
187                 value = max(value, minimax(board, player, depth-1, alpha, beta,
188                 False)*score_matrix[col])
189                 #print("maximizing value: ", value)
190                 alpha = max(alpha, value)
191                 board[row][col] = 0
192                 if value >= beta:
193                     break
194             database[str(board)] = value
195         return value
196     else:
197         value = float('inf')
198         for col in range(7):
199             if is_valid_move(board, col):

```

```

194         row = get_row(board, col)
195         board[row][col] = opponent
196         value = min(value, minimax(board, player, depth-1, alpha, beta,
197             True)*score_matrix[col])
198         #print("minimizing value: ", value)
199         beta = min(beta, value)
200         board[row][col] = 0
201         if value <= alpha:
202             break
203     database[str(board)] = value
204
205 def ai_student(board, player):
206     #print("AI turn :))")
207     opponent = 1 if player == 2 else 2
208     # listing all the legit move
209     legit_moves = [col for col in range(7) if is_valid_move(board, col)]
210     random.shuffle(legit_moves) # Shuffle for random move selection
211     best_score = -float('inf')
212     chosen_col = -1
213     for col in legit_moves:
214         row = get_row(board, col)
215         board[row][col] = player
216         if str(board) in database:
217             score = database.get(str(board))
218         else :
219             score = minimax(board, player, 3, -float('inf'), float('inf'), False)
220         board[row][col] = 0
221         #print(f'Move at col {col} with score {score}')
222         if score > best_score:
223             best_score = score
224             chosen_col = col
225     #print('Chosen col: ', chosen_col, ' with score: ', best_score)
226     return chosen_col

```

## Inputs

- `board` est un tableau (array) numpy de taille  $6 \times 7$  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 indexées 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 :

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

- `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 a choisi de jouer. *Exemple :* 2 (la troisième colonne)

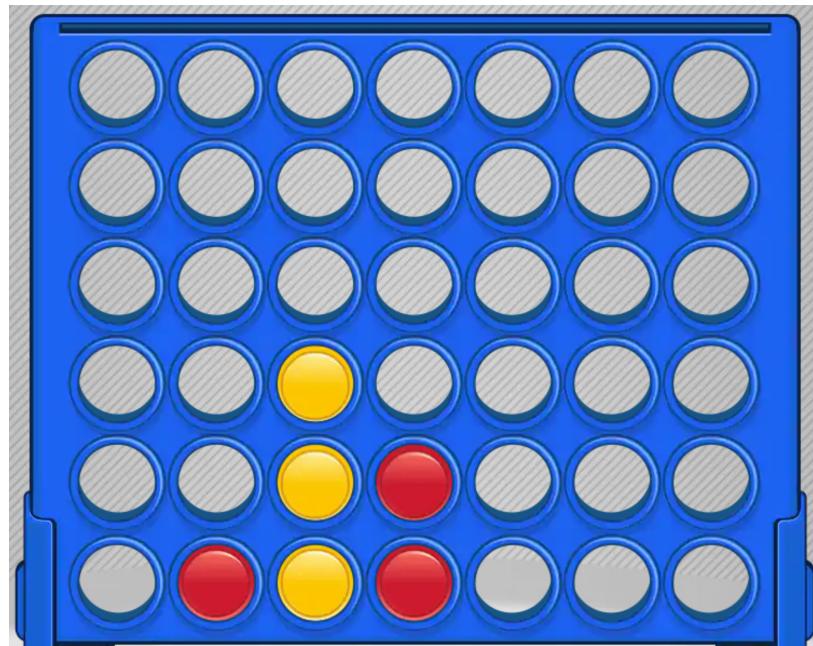


FIGURE 2 – Illustration de l'exemple utilisé pour expliquer le code `ai_student`. Le joueur 1 est représenté par la couleur rouge et le joueur 2 est représenté par la couleur jaune. Le joueur 1 doit jouer la colonne 3 pour éviter de perdre la partie.

## 4 Monte Carlo avec votre intelligence artificielle

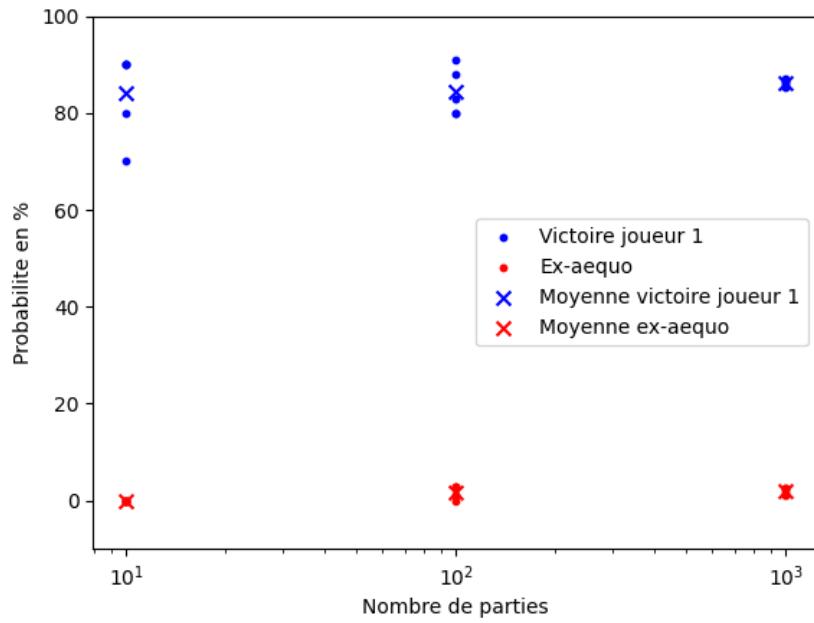
Si vous arrivez à cette partie du devoir, c'est que votre IA a (partiellement) réussi les tests sur Gradescope. Bravo ! Pour observer l'amélioration que votre IA apporte par rapport à l'IA aléatoire évaluée dans la section 2, nous vous proposons de faire jouer l'une contre l'autre. Pour ce faire, nous 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:** Moyenne obtenue pour 5 répétitions de 10, 100 et 1000 parties lorsque l'IA joue en 1ère position : 85%.

Note : Mon IA prend trop de temps pour 10 000 parties, je n'ai pas pu calculer la moyenne pour 10 000 parties car cela me prenait 13 heures, voire plus, pour calculer. J'ai donc réduit le nombre de simulations pour pouvoir calculer la moyenne. Celle-ci n'est donc pas totalement représentative.



Cas 2 : Le joueur 1 joue ai\_random. et le joueur 2 joue ai\_student.

**Solution:** La moyenne obtenue pour le joueur 1 pour 5 répétitions de 10, 100 et 1000 parties lorsque l'IA joue en 2e position : 17%

