

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.

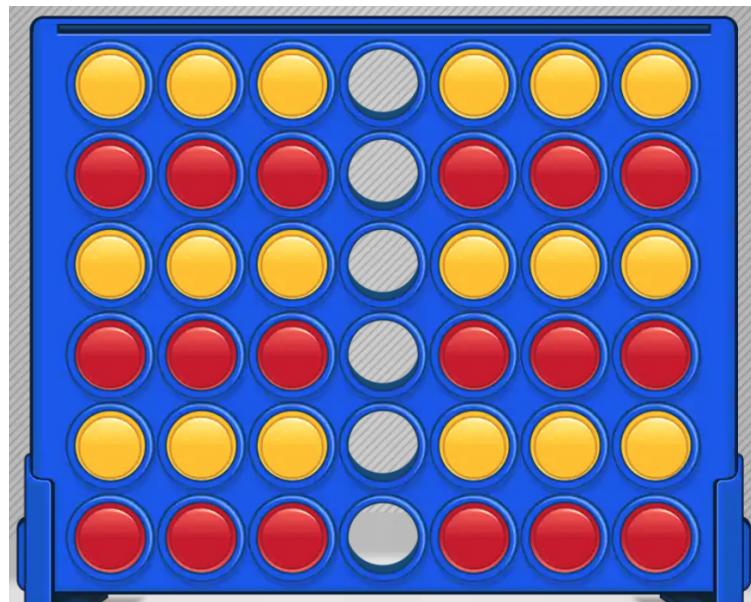


FIGURE 1 – Plateau de jeu considéré pour le calcul de borne inférieure

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 :

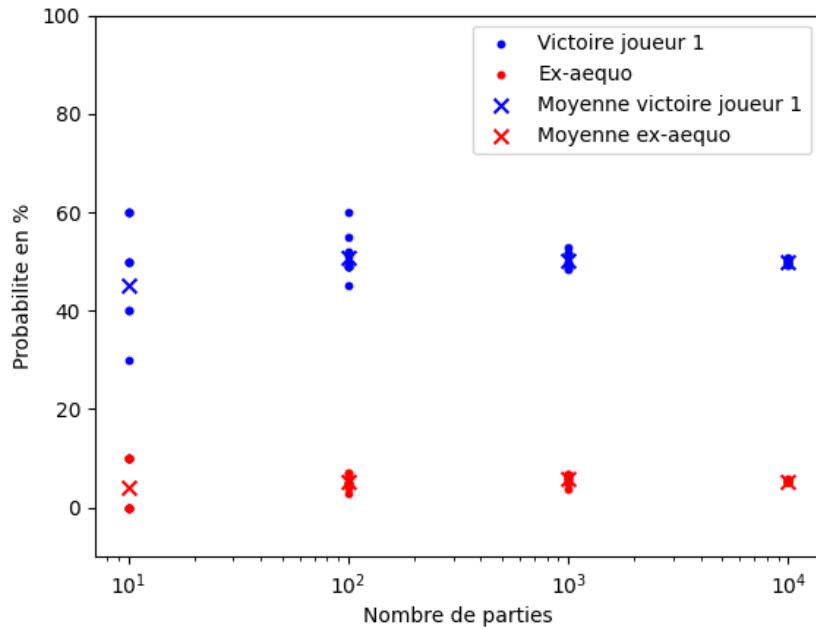
- `connect4.py` : fichier qui réunit toutes les fonctions nécessaires afin de simuler une partie de Puissance 4 :
 - `ai_random` : la stratégie basique considérée dans cette section.
 - `update_board` : met à jour le tableau de jeu à l'aide du coup reçu en argument.
 - `print_board` : permet une visualisation 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 ~ 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 1 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         # undo the move
24         board[row][move] = 0
25         return True
26     # undo the move
27     board[row][move] = 0
28     return False
29
30 def check_win(board, player):
31     # Check horizontal
32     for col in range(COLUMN_COUNT):
33         for row in range(ROW_COUNT):
34             if col+3 < 7:
35                 if board[row][col] == player and board[row][col+1] == player and
board[row][col+2] == player and board[row][col+3] == player:
36                     return True
37
38     # Check vertical
39     if row+3 < 6:
40         if board[row][col] == player and board[row+1][col] == player and
board[row+2][col] == player and board[row+3][col] == player:
41             return True
```

```

42
43     # Check down right diagonals
44     if col+3 < 7 and row+3 < 6:
45         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:
46             return True
47
48     # Check down left diagonals
49     if col-3 >= 0 and row+3 < 6:
50         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:
51             return True
52
53 def evaluate_board(board, player):
54     score = 0
55     opponent = 1 if player == 2 else 2
56     for i in range(four_in_a_row(board, player)):
57         score += 1000
58     for i in range(four_in_a_row(board, opponent)):
59         score -= 1000
60     for i in range(three_in_a_row(board, player)):
61         score += 100
62     for i in range(three_in_a_row(board, opponent)):
63         score -= 100
64     for i in range(two_in_a_row(board, player)):
65         score += 10
66     for i in range(two_in_a_row(board, opponent)):
67         score -= 10
68     for i in range(one_in_a_row(board, player)):
69         score += 1
70     for i in range(one_in_a_row(board, opponent)):
71         score -= 1
72     return score
73
74 def four_in_a_row(board, player):
75     # Check horizontal
76     score = 0
77     for col in range(COLUMN_COUNT):
78         for row in range(ROW_COUNT):
79             if col+3 < 7:
80                 if board[row][col] == player and board[row][col+1] == player and
board[row][col+2] == player and board[row][col+3] == player:
81                     score += 1
82
83         # Check vertical
84         if row+3 < 6:
85             if board[row][col] == player and board[row+1][col] == player and
board[row+2][col] == player and board[row+3][col] == player:
86                 score += 1
87
88         # Check down right diagonals
89         if col+3 < 7 and row+3 < 6:
90             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:
91                 score += 1
92
93     # Check down left diagonals

```

```

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

```

```

142
143 def one_in_a_row(board, player):
144     score = 0
145     for col in range(7):
146         for row in range(6):
147             # checking horizontal
148             if col+3 < 7:
149                 if board[row][col] == player and board[row][col+1] == 0 and board[
150                     row][col+2] == 0 and board[row][col+3] == 0:
151                         score += 1
152             # checking vertical
153             if row+3 < 6:
154                 if board[row][col] == player and board[row+1][col] == 0 and board[
155                     row+2][col] == 0 and board[row+3][col] == 0:
156                     score += 1
157             # checking down left diag
158             if col-3 >= 0 and row+3 < 6:
159                 if board[row][col] == player and board[row+1][col-1] == 0 and
160                     board[row+2][col-2] == 0 and board[row+3][col-3] == 0:
161                         score += 1
162             # checking down right diag
163             if col+3 < 7 and row+3 < 6:
164                 if board[row][col] == player and board[row+1][col+1] == 0 and
165                     board[row+2][col+2] == 0 and board[row+3][col+3] == 0:
166                         score += 1
167     return score
168
169 def is_terminal_state(board):
170     # checking if a player won
171     if four_in_a_row(board, 1) or four_in_a_row(board, 2):
172         return True
173     # checking if board is full
174     for col in range(7):
175         if board[0][col] == 0:
176             return False
177     return True
178
179 score_matrix = [0.2, 0.5, 1.0, 3, 1.0, 0.5, 0.2]
180
181 def minimax(board, player, depth, alpha, beta, maximizingPlayer):
182     opponent = 1 if player == 2 else 2
183     if str(board) in database:
184         return database[str(board)]
185     if depth == 0 or is_terminal_state(board):
186         #print("terminal state")
187         return evaluate_board(board, player)
188
189     if maximizingPlayer:
190         value = -float('inf')
191         for col in range(7):
192             if is_valid_move(board, col):
193                 row = get_row(board, col)
194                 board[row][col] = player
195                 value = max(value, minimax(board, player, depth-1, alpha, beta,
196                 False)*score_matrix[col])
197                 #print("maximizing value: ", value)
198                 alpha = max(alpha, value)
199
200

```

```

194         board[row][col] = 0
195         if value >= beta:
196             break
197     database[str(board)] = value
198     return value
199 else:
200     value = float('inf')
201     for col in range(7):
202         if is_valid_move(board, col):
203             row = get_row(board, col)
204             board[row][col] = opponent
205             value = min(value, minimax(board, player, depth-1, alpha, beta,
True)*score_matrix[col])
206             #print("minimizing value: ", value)
207             beta = min(beta, value)
208             board[row][col] = 0
209             if value <= alpha:
210                 break
211     database[str(board)] = value
212     return value
213
214 def ai_student(board, player):
215     #print("AI turn :))")
216     opponent = 1 if player == 2 else 2
217     # listing all the legit move
218     legit_moves = [col for col in range(7) if is_valid_move(board, col)]
219     random.shuffle(legit_moves) # Shuffle for random move selection among equally
good moves
220
221     best_score = -float('inf')
222     chosen_col = -1
223     for col in legit_moves:
224         row = get_row(board, col)
225         board[row][col] = player
226         if str(board) in database:
227             score = database.get(str(board))
228         else :
229             score = minimax(board, player, 3, -float('inf'), float('inf'), False)
230         board[row][col] = 0
231         #print(f'Move at col {col} with score {score}')
232         if score > best_score:
233             best_score = score
234             chosen_col = col
235
236     #print('Chosen col: ', chosen_col, ' with score: ', best_score)
237     return chosen_col

```

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 :

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

- `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)

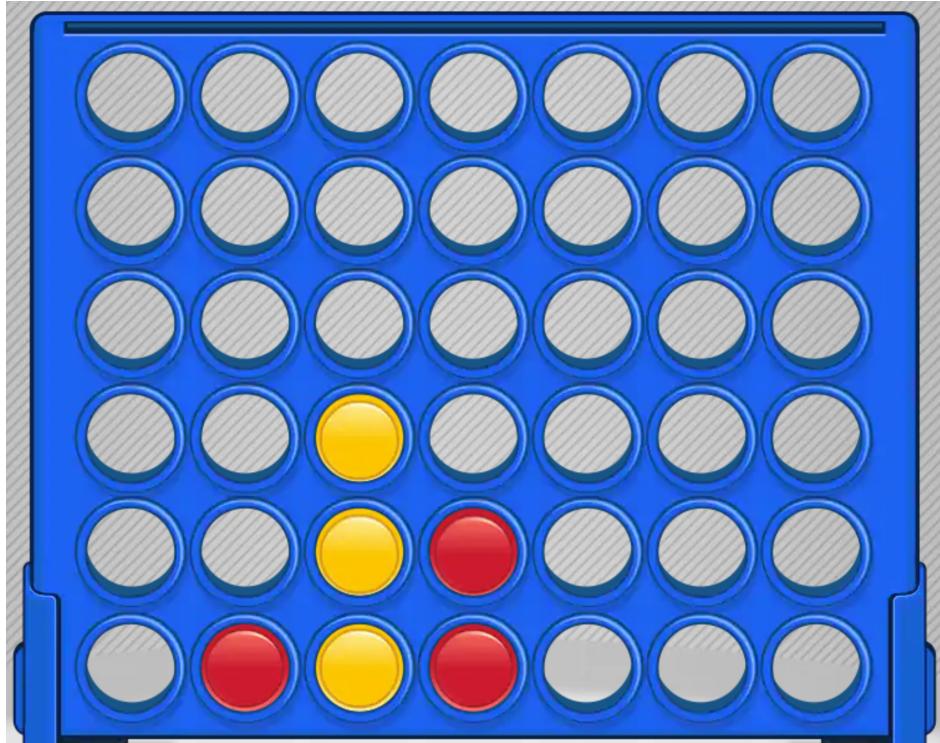


FIGURE 2 – Illustration de l'exemple utilisé pour expliquer le code `ai_student`. Le joueur 1 est représenté par la couleur rouge et le joueur 2 est représenté par la couleur jaune. Le joueur 1 doit jouer la colonne 3 pour éviter de perdre la partie.

4 Monte Carlo avec votre intelligence artificielle

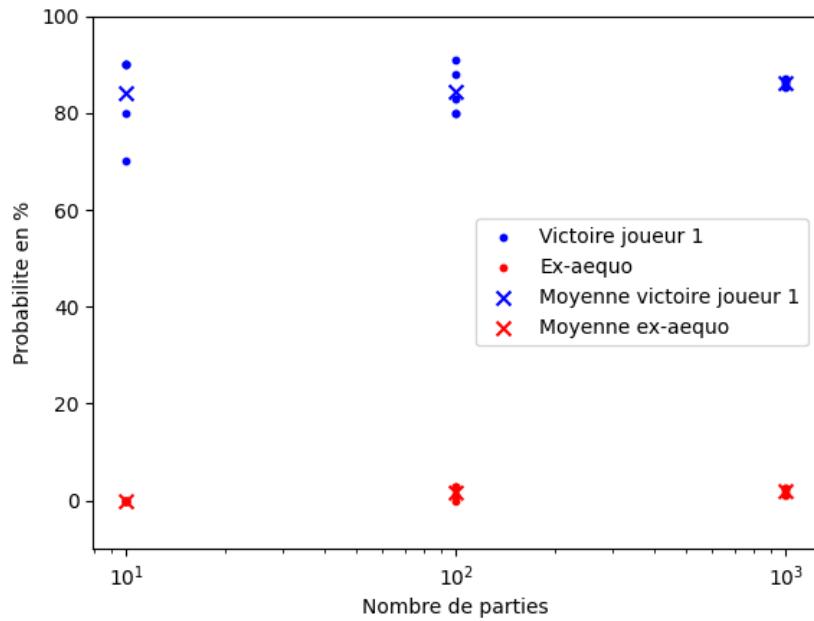
Si vous arrivez à cette partie du devoir, c'est que votre IA a (partiellement) réussi les tests sur Gradescope. Bravo ! Pour observer l'amélioration que votre IA apporte par rapport à l'IA aléatoire évaluée dans la section 2, nous vous proposons de faire jouer l'une contre l'autre. Pour ce faire, nous 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%

