Programmation du jeu Hexapawn en Python

13 avril 2022

Table des matières

1 Programmation du jeu Hexapawn

1.1 Appels des modules

Exécuter la cellule ci-dessous afin d'importer les modules et fonctions nécessaires au bon fonctionnement des programmes.

[ ]:
```
import copy
import random
import matplotlib.pyplot as plt
from hexapawn_fcts import *
```

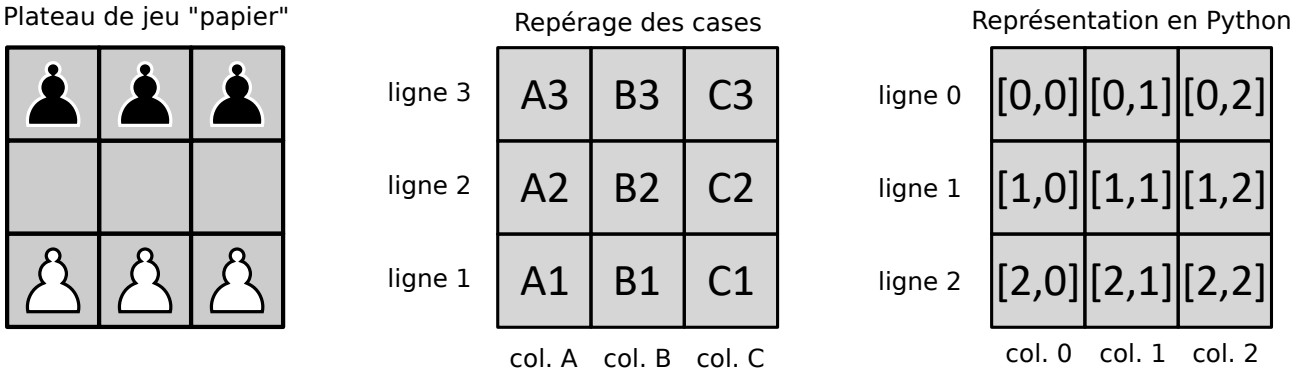
1.2 Modélisation du plateau de jeu et des mouvements

1.2.1 Plateau de jeu

Le plateau de jeu d'Hexapawn peut être considéré comme un tableau de 3 lignes et de 3 colonnes. Chaque case peut-être repérée par son numéro de ligne et son numéro de colonne.

En Python, nous allons représenter le plateau sous la forme d'une liste de listes, de sorte que le plateau initial sera représenté par la liste : `[["N","N","N"],[" "," "," "],["B","B","B"]]`, un pion blanc étant représenté par le caractère `B`, un pion noir par le caractère `N` et une case vide par une espace.

Les positions des pions seront exprimées sous forme de couples (listes de deux entiers), comme des coordonnées. Les mouvements seront alors représentés par des couples de couples `[coordonnées de départ, coordonnées d'arrivée]`.



Cette structure de données de type `list` va nous permettre de manipuler les éléments du plateau : déplacement et suppression seront réalisés par de simples affectations d'un caractère au bon emplacement.

Afin de faciliter la visualisation d'un plateau, il est possible d'obtenir un rendu en construisant une fonction d'affichage, nommée `affichage`.

Vous pouvez exécuter la cellule ci-dessous pour obtenir l'affichage du plateau.

[ ]:
```
exemple = [["N"," ","N"],[" ","N","B"],["B","B"," "]];
affichage(exemple) # exécutez la cellule pour avoir le rendu
```

1.2.2 Etats de jeux et fonctions associées

Plusieurs fonctions auxiliaires sont nécessaires pour faire fonctionner le jeu. Pour commencer, nous avons besoin d'une fonction `positions` donnant les positions des pions pour une couleur donnée :

Vérifions ce que fait cette fonction avec un exemple. Exécutez la cellule ci-dessous :

[ ]:
```
exemple = [["N"," ","N"],[" ","N","B"],["B","B"," "]];
affichage(exemple);
positions(exemple,"N") # les valeurs renvoyées sont les coordonnées des␣
 ↪pions noirs
```

Nous avons ensuite besoin d'une fonction `coups_possibles` qui détermine les mouvements possibles pour les pions d'une couleur lorsque c'est à son tour de jouer :

Vérifions ce que fait cette fonction sur l'exemple précédent :

[ ]:
```
exemple = [["N"," ","N"],[" ","N","B"],["B","B"," "]];
affichage(exemple);
coups_possibles(exemple,"N")
```

Nous pouvons ensuite construire une fonction `plateau_joue` qui va donner le résultat d'un mouvement à partir d'un état de jeu :

Pour l'exemple, nous pouvons par exemple faire jouer les noirs de B2 vers A1 (mouvement `[[1, 1], [2, 0]]` en coordonnées), ce qui donne :

[ ]:
```
exemple = [["N"," ","N"],[" ","N","B"],["B","B"," "]];
affichage(exemple);
mouvement = [[1, 1], [2, 0]];
exemple_apres_coup = plateau_joue(exemple, "N", mouvement);
affichage(exemple_apres_coup)
```

Nous pouvons donc désormais réaliser les mouvements, modifier l'état des plateaux. Il nous reste à pouvoir tester si une partie est terminée ou non, après le coup joué par une couleur. On crée une fonction `fin_de_partie`.

En reprenant l'exemple précédent, nous avons joué un pion noir qui a atteint la ligne des blancs donc il devrait y avoir une victoire (par conquête).

[ ]:
```
fin_de_partie(exemple_apres_coup, "N") # Victoire des noirs ?
```

1.3 Jeu entre deux joueurs aléatoires

1.3.1 Fonction `partie_alea_vs_alea`

Rappel : selon l'arbre de jeu établi en classe, nous avons en théorie, lors d'une partie aléatoire, une probabilité de $\frac{259}{432} \approx 0,60$ de victoire pour les blancs et une probabilité de $\frac{173}{432} \approx 0,40$ pour les noirs.

On va construire une fonction de jeu simulant une partie entre deux joueurs aléatoires dont les coups seront choisis au hasard grâce à une fonction aléatoire de Python. Une fois la fonction de jeu construite, il sera facile de simuler un grand nombre de parties par répétition à l'aide d'une boucle.

La fonction `partie_alea_vs_alea()` va se construire en suivant la démarche suivante :

- on part du plateau initial `[["N","N","N"],[" "," "," "],["B","B","B"]]` et on crée une variable `tour` initialisée à 0 qui va indiquer quelle couleur doit jouer. Les blancs, **qui commencent toujours**, auront donc les tours pairs (0, 2, 4, 6) et les noirs les tours impairs (1, 3, 5, 7).
- on crée une variable `vainqueur`qui renverra la couleur gagnante : `"B"`ou `"N"`
- on crée une variable `partie_en_cours` qui indiquera à chaque fois si la partie se poursuit ou non, après vérification d'une éventuelle victoire par la fonction `fin_de_partie`. La variable `partie_en_cours` est initialisée à `True` et prendra la valeur `False` dès qu'un vainqueur aura été désigné, ce qui arrêtera la partie (ce type de variable s'appelle un **booléen**).
- tant que `fin_de_partie` est à la valeur `True`, cela signifie qu'on continue à jouer de la manière suivante :
  - à chaque tour, on choisit un coup au hasard avec la fonction de choix aléatoire `choice` du module `random` dans la liste des coups possibles : on utilise la fonction `coups_possibles`
  - on joue ce coup et on obtient un nouvel état du plateau : cela utilise la fonction `coup_joue`
  - on teste si le nouvel état de jeu a amené la victoire de la couleur qui vient de jouer : on utilise la fonction `fin_de_partie` pour tester une éventuelle situation de victoire
  - s'il y a victoire, on met la variable `partie_en_cours` à `False`, ce qui arrête la partie
  - sinon on ne fait rien et on poursuit la répétition, avec l'affectation `tour = tour + 1`
- quand la partie est terminée, la variable `vainqueur` contient la couleur qui vient de gagner et on la renvoie.

La fonction `partie_alea_vs_alea` a été partiellement construite : les instructions ont été saisies pour le tour des blancs.

Par analogie, compléter les instructions à saisir pour le tour des noirs.

```
[ ]: import random
def partie_alea_vs_alea():
    """renvoie une partie aléatoire entre deux joueurs qui jouent au␣
 ↪hasard"""
    tour = 0
    plateau = [["N","N","N"],[" "," "," "],["B","B","B"]]
    vainqueur = ""
    partie_en_cours = True
    while partie_en_cours == True:
        if tour % 2 == 0: # au tour des blancs de jouer
            coup_choisi = random.choice(coups_possibles(plateau,"B")) #␣
 ↪choix d'un coup au hasard
            plateau = plateau_joue(plateau, 'B', coup_choisi) # coup joué
            if fin_de_partie(plateau, "B") == True: # vérification d'une␣
 ↪éventuelle victoire des blancs
                partie_en_cours = False # arrêt de la partie
                vainqueur = "B"
        if tour % 2 == 1: # au tour des noirs de jouer
            ...
            ...
            ...
            ...
            ...
        tour = tour + 1 # on passe au tour suivant
    pass #return vainqueur # remplacer pass par l'instruction mise en␣
 ↪commentaire
```

Tester votre fonction en exécutant plusieurs fois de suite la cellule ci-dessous.

```
[ ]: partie_alea_vs_alea() # appel de la fonction pour réaliser une partie␣
 ↪aléatoire
```

1.3.2 Simulation de parties

Afin de vérifier que les victoires se répartissent bien selon la probabilité théorique (60/40), nous allons construire un échantillon qui va réaliser `nb_parties` parties et compter les nombres de victoires de chaque couleur.

Compléter la fonction `echantillon(nb_parties)` en saisissant les instructions suivantes :

— déclarer une variable `victoires_blancs`, initialisée à 0 et qui comptera les victoires des blancs
— déclarer une variable `victoires_noirs`, initialisée à 0 et qui comptera les victoires des noirs

— réaliser une boucle de longueur `nb_parties` pour répéter `nb_parties` fois les instructions suivantes :

  — réaliser une partie aléatoire en appelant la fonction précédente. Vous affecterez le résultat de cet appel dans une variable `partie`
  — selon la valeur de `partie` (`"B"` ou `"N"`), augmenter le compteur correspondant d'une unité

— renvoyer la fréquence des victoires de chaque couleur

[ ]:
```
def echantillon(nb_parties):
    """renvoie la fréquence de victoire de chaque couleur après␣
↪simulation de nb_parties parties"""
    victoires_blancs = 0
    victoires_noirs = 0
    for _ in range(nb_parties):
        ...
        ...
        ...
        ...
    pass #return victoires_blancs / nb_parties, victoires_noirs /␣
↪nb_parties # remplacer pass par l'instruction mise en commentaire
```

Tester votre fonction en exécutant la cellule ci-dessous.

[ ]:
```
echantillon(1000) #
```

1.3.3 Représentation graphique

— on **monte** d'une unité en diagonale en cas de victoire des **noirs** ;
— on **descend** d'une unité en diagonale en cas de victoire des **blancs**

Afficher la trajectoire obtenue en faisant 100 parties.

[ ]:
```
graphique([partie_alea_vs_alea() for _ in range(100)])
```

1.4 Jeu entre un joueur aléatoire et une "IA"

1.4.1 Retour sur le jeu de 1962

Historiquement, le jeu d'Hexapawn a été créé par Martin Gardner pour simuler une intelligence artificielle en papier, composée de 24 boîtes d'allumettes contenant des perles de couleur.

Chaque boîte correspond à l'une des situations du plateau de jeu (ou son symétrique) et les couleurs aux différents coups que la machine peut réaliser dans cette situation.

Dans sa configuration initiale (celle que nous avons modélisée dans le tableur), lorsque c'est au tour de la machine de jouer, elle tire une perle au hasard dans la boîte correspondant à la situation et qui lui indique le coup qu'elle doit jouer. Cette perle est mise de côté et, à la fin de la partie, la règle est la suivante :

— si la machine a gagné, toutes les perles sont remises dans leurs boîtes ;
— si elle a perdu, la perle qui l'a fait perdre est éliminée et le coup associé ne peut plus être joué

En multipliant les parties, le retrait progressif des perles perdantes va permettre à la machine d'apprendre les bons coups en élaguant l'arbre de jeu pour ne conserver que les branches gagnantes.

Cette méthode d'apprentissage se base sur un système de *punition* : si l'IA perd, on la *punit* de façon à ce qu'elle ne reproduise plus l'erreur qu'elle a commise pour perdre. À force elle ne commet plus d'erreur.

Mais on peut aussi envisager de procéder par *récompense*. Lorsque que l'IA gagne, on rajoute une ou plusieurs perles de la couleur du coup gagnant dans la dernière boîte. Ainsi, l'IA a plus de chance de reproduire ce coup gagnant plus tard car la probabilité de tomber sur cette perle gagnante est augmentée.

Dans les séances précédentes, nous avons simulé avec le tableur le fonctionnement d'une machine qui joue des coups au hasard contre un humain et qui élimine les coups perdants pour elle. Nous allons simuler ce fonctionnement et le généraliser en créant plusieurs objets :

— création de l'arbre de jeu (que nous avions dans le fichier tableur) sous forme de listes
— création d'une pioche contenant tous les états possibles du jeu et tous les coups possibles

1.4.2 Création de l'arbre de jeu

Exécutez la cellule-ci dessous pour voir ce que produit la fonction `arbre de jeu`

[ ]:
```
print(arbre_de_jeu()[37]); # affiche un exemple de partie pris dans␣
↪l'arbre de jeu
len(arbre_de_jeu()) # affiche le nombres de parties possibles
```

1.4.3 Création de la pioche et d'une fonction de recherche dans celle-ci

Nous créons ensuite une pioche (fonction `boites`) qui va contenir les états de jeu et les coups possibles associés. À l'instar du jeu en version papier qui utilisait des boites, nous pouvons imaginer chaque étape du jeu comme une liste Python qui contient la situation du plateau et les coups possibles qui peuvent être joués pour la couleur qui doit jouer.

Pour pouvoir nous repérer dans les différentes listes, nous créons une fonction de recherche `recherche` afin de retrouver la partie qui se joue et les coups possibles.

Exécutez la cellule-ci dessous pour comprendre comment fonctionnent ces deux fonctions

[ ]:
```
boites()[3][4][0] # pour un tour des noirs (rang 3, impair), situation␣
↪de jeu numérotée 4 dans la pioche
```

[ ]:
```
recherche([['N', 'N', ' '], ['B', 'B', 'N'], [' ', ' ', 'B']],3) # retrouve le␣
↪numéro de la situation qui correspond au plateau
```

1.4.4 Création de la fonction de jeu

Comme évoqué plus haut, le principe de l'apprentissage par renforcement peut être décliné selon un système de récompense et/ou de punition. Ainsi la fonction `sequence_alea_vs_ia` prend en argument un nombre de parties `nb_parties` et va faire jouer un joueur aléatoire contre la machine qui va se comporter de manière différente selon les paramètres optionnels `punition` et `recompense`.

[ ]:
```
def sequence_alea_vs_ia(nb_parties, punition = True, recompense = False):
    """renvoie une serie de nb_parties parties d'hexapawn entre un joueur␣
↪aléatoire et une IA qui valorise ou élimine ses coups selon l'issue de␣
↪la partie et le type de feedback choisi"""
    pioche = boites() # etats de jeux avec plateaux et coups possibles
    liste_vainqueurs = []
    liste_parcours = []
    for _ in range(nb_parties): # on simule n parties et on élimine les␣
↪parties perdantes pour l'IA
```

```
        tour = 0
        plateau = [["N","N","N"],[" "," "," "],["B","B","B"]]
        partie_en_cours = True
        dernier_coup_noirs = []
        dernier_coup_noirs_sym = []
        while partie_en_cours == True:
            if tour % 2 == 0: # au tour des blancs de jouer
                coup = random.choice(coups_possibles(plateau,"B")) #␣
↪choix d'un coup au hasard
                plateau = plateau_joue(plateau, 'B', coup) # coup joué
                if fin_de_partie(plateau, "B") == True: # vérification␣
↪d'une éventuelle victoire des blancs
                    partie_en_cours = False # arrêt de la partie
                    liste_vainqueurs = liste_vainqueurs + ["B"]
                    if punition:
                        rang, partie, mouvement = dernier_coup_noirs.
↪pop() # rappel du dernier coup noir
                        rang_sym, partie_sym, mouvement_sym =␣
↪dernier_coup_noirs_sym.pop()  # et de son symétrique
                        pioche[rang][partie][1].remove(mouvement) #␣
↪suppression de ce choix perdant pour l'IA (punition)
                        pioche[rang_sym][partie_sym][1].
↪remove(mouvement_sym)
            if tour % 2 == 1: # au tour des noirs de jouer IA
                partie = recherche(plateau,tour) # recherche de la partie␣
↪dans les états de jeux
                partie_symetrique = recherche(sym_plateau(plateau),tour)␣
↪# recherche de la partie symétrique
                if pioche[tour][partie][1] == []: # "boite" vide donc les␣
↪noirs ne peuvent pas jouer et on recherche le dernier mouvement noir␣
↪à éliminer
                    partie_en_cours = False # arrêt de la partie
                    liste_vainqueurs = liste_vainqueurs + ["B"] # ajout␣
↪du vainqueur blanc à la liste des vainqueurs
                    if punition:
                        rang, partie, mouvement = dernier_coup_noirs.pop()
                        rang_sym, partie_sym, mouvement_sym =␣
↪dernier_coup_noirs_sym.pop()
                        pioche[rang][partie][1].remove(mouvement) #␣
↪suppression du dernier mouvement
                        pioche[rang_sym][partie_sym][1].
↪remove(mouvement_sym)
                else:
```

```
                    coup = random.choice(pioche[tour][partie][1]) # l'IA␣
↪choisit un coup au hasard dans la boite correpondant à l'état de jeu
                    dernier_coup_noirs.append([tour, partie, coup])  # on␣
↪enregistre le dernier coup noir (fonctionnement comme une pile, ici␣
↪on empile
                    dernier_coup_noirs_sym.append([tour,␣
↪partie_symetrique, sym_mouv(coup)])
                    plateau = plateau_joue(plateau, 'N', coup)
                    if fin_de_partie(plateau, "N") == True: # victoire de␣
↪la machine
                        partie_en_cours = False
                        liste_vainqueurs = liste_vainqueurs + ["N"]
                        if recompense: # on recompense le dernier␣
↪mouvement gagnant en le rajoutant plusieurs fois dans la pioche afin␣
↪d'augmenter sa probabilité d'être choisi aux prochains tours
                            rajout = 3 # le nombre de coups gagnants␣
↪rajouté peut être modifié ici
                            for _ in range(rajout):
                                pioche[tour][partie][1].append(coup)
                                pioche[tour][partie_symetrique][1].
↪append(sym_mouv(coup))
            tour = tour + 1
    return liste_vainqueurs
```

Exécutez la cellule-ci dessous pour voir ce que produit la fonction `sequence_alea_vs_ia`**.**

[ ]:
```
sequence_alea_vs_ia(100)
```

1.4.5 Visualisations graphiques

Afficher le graphique pour la liste `sequence_alea_vs_ia(100, punition = False, recompense = False)`**.** Que retrouve-t-on ?

[ ]:
```

```

Afficher le graphique pour une séquence de 50 parties, sans punition mais avec récompense. Comment jugez-vous l'apprentissage de la machine ? (Afin de vous faire une idée, n'hésitez pas à ré-exécuter la cellule afin de relancer la séquence)

Réessayez avec un nombre de parties plus important.

[ ]:
```

```

Afficher le graphique pour une séquence de 50 parties, avec punition mais sans récompense. Comment jugez-vous l'apprentissage de la machine ? (Afin de vous faire une idée, n'hésitez

pas à ré-exécuter la cellule afin de relancer la séquence)

Tester avec 200 parties.

[ ]:

Afficher le graphique pour une séquence de 50 parties, avec punition et avec récompense. Comment jugez-vous l'apprentissage de la machine ? (Afin de vous faire une idée, n'hésitez pas à ré-exécuter la cellule afin de relancer la séquence)

Tester avec 200 parties.

[ ]:

1.5 Recherche d'une stratégie gagnante

Les méthodes d'apprentissage vues plus haut montrent que l'IA devient imbattable au bout d'un certain nombre de parties, ce qui signifie qu'**il existe une stratégie gagnante pour les noirs**.

Nous allons reprendre la fonction précédente et constituer la liste des parties gagnées par l'IA afin de voir si une stratégie se dégage.

Exécutez la cellule ci-dessous. Qu'affiche-t-elle ?

[ ]:
```
serie = recherche_strategie(5000, punition = True, recompense = False)
for partie_gagnante in serie[0]:
    print(partie_gagnante)
print(serie[1])
```

Exécutez la cellule ci-dessous. Qu'affiche-t-elle ? Quelle est la différence avec la série précédente ?

[ ]:
```
serie_bis = recherche_strategie(5000, punition = True, recompense = True)
for partie_gagnante in serie_bis[0]:
    print(partie_gagnante)
print(serie_bis[1])
```

À l'aide des résultats, rédiger un texte qui décrit une stratégie gagnante pour les noirs, quels que soient les coups joués par les blancs. Votre réponse :