# Notebook professeur pour la programmation du jeu Hexapawn

13 avril 2022

## Table des matières

# 1 Programmation du jeu Hexapawn

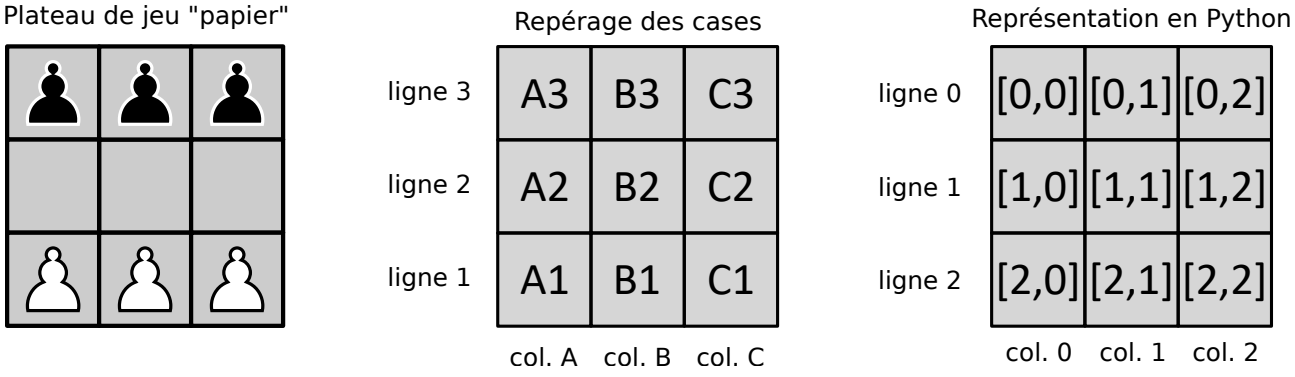
## 1.1 Modélisation du plateau de jeu et des mouvements

### 1.1.1 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`.

[87]:
```
import copy
def affichage(plateau):
    hrule = "   "+'-' * 13
    plateau_aff = copy.deepcopy(plateau) + [["A","B","C"]]
    print(hrule)
    for i in range(len(plateau_aff)):
        ligne = plateau_aff[i]
        if i < 3:
            print(str(3-i)+" ",'|', ' | '.join(cell for cell in ligne), '|')
            print(hrule)
        else:
```

```
            print("  ",' ', '   '.join(cell for cell in ligne), '')
```

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

[88]:
```
exemple = [["N"," ","N"],[" ","N","B"],["B","B"," "]];
affichage(exemple) # exécutez la cellule pour avoir le rendu
```

```
   -------------
3  | N |   | N |
   -------------
2  |   | N | B |
   -------------
1  | B | B |   |
   -------------
     A   B   C
```

### 1.1.2 États de jeux et fonctions associées

Plusieurs fonctions auxiliaires sont nécessaires pour faire fonctionner le jeu. Il n'est pas essentiel de s'y attarder lors d'une première lecture.

[89]:
```
def coord_vers_chaine(couple):
    """convertit un mouvement d'une liste de deux listes vers une chaine␣
 ↪de caractères"""
    cases = [['A3', 'B3', 'C3'], ['A2', 'B2', 'C2'], ['A1', 'B1', 'C1']]
    couples = [[[0,0], [0,1], [0,2]], [[1,0], [1,1],[1,2]], [[2,0],␣
 ↪[2,1], [2,2]]]
    conversion = cases[couple[0][0]][couple[0][1]] +␣
 ↪cases[couple[1][0]][couple[1][1]]
    return conversion

def chaine_vers_coord(chaine):
    """convertit un mouvement sous la forme d'une chaine de caractères␣
 ↪vers une liste de deux listes"""
    abs = []
    ord = []
    cases = [['A3', 'B3', 'C3'], ['A2', 'B2', 'C2'], ['A1', 'B1', 'C1']]
    couples = [[[0,0], [0,1], [0,2]], [[1,0], [1,1],[1,2]], [[2,0],␣
 ↪[2,1], [2,2]]]
    for i in range(3):
        for j in range(3):
            if cases[i][j] == chaine[:2]:
                abs = couples[i][j]
            if cases[i][j] == chaine[2:]:
                ord = couples[i][j]
```

```
    return [abs,ord]


def direction(couleur):
    """direction de mouvement selon la couleur"""
    direction = -1
    if couleur == "N":
        direction = 1
    elif couleur == "B":
        direction = -1
    return direction

def autre_couleur(couleur):
    """renvoie l'autre couleur"""
    if couleur == "N":
        return "B"
    elif couleur == "B":
        return "N"

def rang_initial(couleur):
    """renvoie le rang initial de la couleur"""
    rang = 1
    if couleur == "N":
        rang = 0
    elif couleur == "B":
        rang = 2
    return rang


def sym_mouv(mouvement):
    """renvoie le symétrique d'un mouvement sous la forme d'un couple"""
    mouv = copy.deepcopy(mouvement)
    lg_depart = mouv[0][0]
    col_depart = mouv[0][1]
    lg_arrivee = mouv[1][0]
    col_arrivee = mouv[1][1]
    return [[lg_depart, 2 - col_depart], [lg_arrivee, 2 - col_arrivee]]


def sym_chaine(chaine):
    """renvoie le symétrique d'un mouvement en échangeant A et C"""
    sym = ""
    for lettre in chaine:
        if lettre == 'A':
```

```
            car = 'C'
        elif lettre == 'C':
            car = 'A'
        else:
            car = lettre
        sym = sym + car
    return sym

def sym_plateau(plateau):
    """renvoie le symétrique du plateau de jeu par rapport à l'axe␣
 ↪vertical"""
    p = copy.deepcopy(plateau)
    for i in range(3):
        temp = p[i][0]
        p[i][0] = p[i][2]
        p[i][2] = temp
    return p

def donne_couleur(rang):
    couleur = ''
    if rang % 2 == 0:
        couleur = 'B'
    elif rang % 2 == 1:
        couleur = 'N'
    return couleur

def etat_plateau(liste_coups):
    """renvoie l'état d'un plateau après la liste des coups joués"""
    plateau = [["N","N","N"],[" "," "," "],["B","B","B"]]
    for i in range(len(liste_coups)):
        coup = chaine_vers_coord(liste_coups[i])
        if i%2 == 0:
            couleur = "B"
        else:
            couleur = "N"
        plateau[coup[0][0]][coup[0][1]] = " "
        plateau[coup[1][0]][coup[1][1]] = couleur
    return plateau
```

Nous avons besoin d'une fonction donnant les positions des pions pour une couleur donnée :

[90]:
```
def positions(plateau, couleur):
    """dresse la liste des positions des pions de la couleur donnée dans␣
 ↪un plateau donné"""
```

```
    liste_positions = []
    for i in range(3):
        for j in range(3):
            if plateau[i][j] == couleur:
                liste_positions = liste_positions + [[i,j]]
    return liste_positions
```

Vérifions ce que fait cette fonction avec un exemple :

[91]:
```
exemple = [["N"," ","N"],[" ","N","B"],["B","B"," "]];
affichage(exemple);
positions(exemple,"N") # les valeurs renvoyés sont les coordonnées des␣
 ↪pions noirs
```

```
   -------------
3  | N |   | N |
   -------------
2  |   | N | B |
   -------------
1  | B | B |   |
   -------------
     A   B   C
```

[91]:
```
[[0, 0], [0, 2], [1, 1]]
```

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

[92]:
```
def coups_possibles(plateau, couleur_a_jouer):
    """dresse la liste des mouvements possibles sur le plateau pour la␣
 ↪couleur qui doit jouer"""
    liste_coups = []
    adversaire = autre_couleur(couleur_a_jouer)
    dir = direction(couleur_a_jouer)
    for pos in positions(plateau, couleur_a_jouer):
        i = pos[0]
        j = pos[1]
        if i != rang_initial(adversaire): # cela veut dire qu'on peut␣
 ↪encore avancer
            if plateau[i+dir][j] == " ":
                liste_coups = liste_coups + [[pos,[i+dir,j]]]
            if j < 2 and plateau[i+dir][j+1] == adversaire:
                liste_coups = liste_coups + [[pos,[i+dir,j+1]]]
            if j > 0 and plateau[i+dir][j-1] == adversaire:
                 liste_coups = liste_coups + [[pos,[i+dir,j-1]]]
```

```
    return sorted(liste_coups, key = lambda couple :␣
↪(couple[0][1],couple[1][1]))
```

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

[93]:
```
exemple = [["N"," ","N"],[" ","N","B"],["B","B"," "]];
affichage(exemple);
coups_possibles(exemple,"N")
```

```
   -------------
3  | N |   | N |
   -------------
2  |   | N | B |
   -------------
1  | B | B |   |
   -------------
     A   B   C
```

[93]:
```
[[[0, 0], [1, 0]], [[1, 1], [2, 0]]]
```

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

[94]:
```
def plateau_joue(plateau, couleur_a_jouer, coup):
    """renvoie le plateau après le coup joué par la couleur"""
    p = copy.deepcopy(plateau)
    if coup in coups_possibles(plateau, couleur_a_jouer):
        p[coup[0][0]][coup[0][1]] = " "
        p[coup[1][0]][coup[1][1]] = couleur_a_jouer
    return p
```

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 :

[95]:
```
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)
```

```
   -------------
3  | N |   | N |
   -------------
2  |   | N | B |
   -------------
1  | B | B |   |
```

```
   -------------
     A   B   C
   -------------
3  | N |   | N |
   -------------
2  |   |   | B |
   -------------
1  | N | B |   |
   -------------
     A   B   C
```

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.

[96]:
```
def fin_de_partie(plateau, couleur_jouee):
    """teste la victoire des pions de la couleur indiquée, en supposant␣
 ↪que ceux-ci viennent de jouer"""
    partie_finie = False
    adversaire = autre_couleur(couleur_jouee)
    if positions(plateau, adversaire) == []:
        partie_finie = True
    for pos in positions(plateau, couleur_jouee):
        i = pos[0]
        j = pos[1]
        if i == rang_initial(adversaire):
            partie_finie = True
    if  len(positions(plateau, adversaire)) > 0 and␣
 ↪len(coups_possibles(plateau, adversaire)) == 0:
        partie_finie = True
    return partie_finie
```

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).

[97]:
```
fin_de_partie(exemple_apres_coup, "N") # Victoire des noirs ?
```

[97]:
```
True
```

## 1.2 Jeu entre deux joueurs aléatoires

### 1.2.1 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.**

[98]:
```
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
            coup_choisi = random.choice(coups_possibles(plateau,"N")) #␣
↪choix d'un coup au hasard
            plateau = plateau_joue(plateau, 'N', coup_choisi) # coup joué
            if fin_de_partie(plateau, "N") == True: # vérification d'une␣
↪éventuelle victoire des noirs
                partie_en_cours = False # arrêt de la partie
                vainqueur = "N"
        tour = tour + 1 # on passe au tour suivant
    return vainqueur
```

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

[99]:
```
partie_alea_vs_alea() # appel de la fonction pour réaliser une partie␣
↪aléatoire
```

[99]: 'N'

### 1.2.2 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

[100]:
```
def echantillon(nb_parties):
    """renvoie la fréquence de victoire de chaque couleur après␣
 ↪simulation de n parties"""
    victoires_blancs = 0
    victoires_noirs = 0
    for _ in range(nb_parties):
        partie = partie_alea_vs_alea()
        if partie == 'B':
            victoires_blancs = victoires_blancs + 1
        if partie == 'N':
            victoires_noirs = victoires_noirs + 1
    return victoires_blancs / nb_parties, victoires_noirs / nb_parties
```

**Tester votre fonction en exécutant la cellule ci-dessous.**

[101]:
```
echantillon(5000) #
```

[101]:
```
(0.5842, 0.4158)
```

### 1.2.3 Représentation graphique

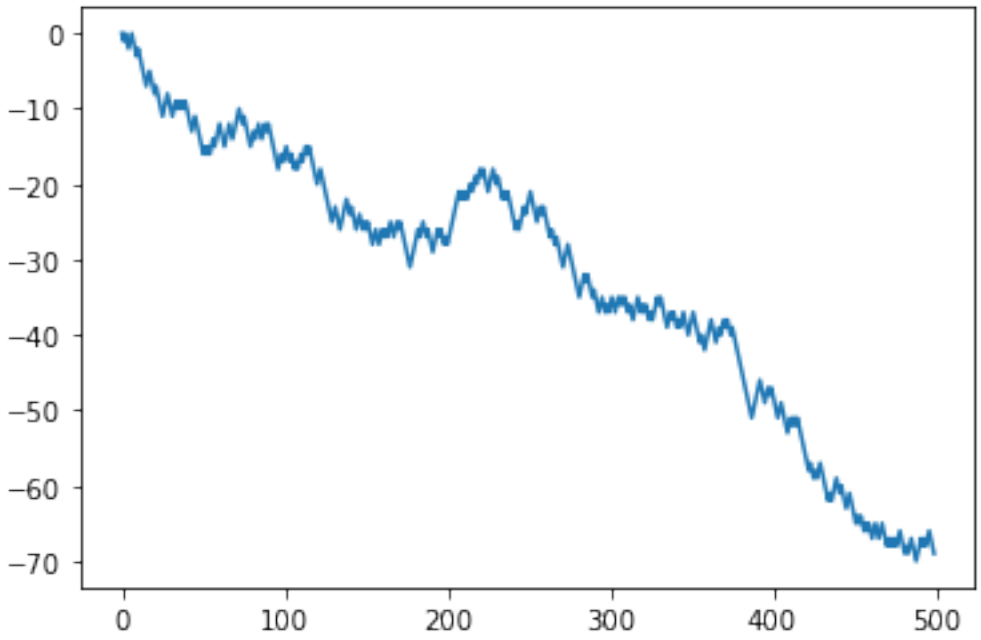
Afin de visualiser l'évolution des victoires, il est possible d'utiliser une fonction `graphique` qui prend en paramètre la liste des vainqueurs et va afficher l'évolution de la trajectoire des victoires :

— 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**

[102]:
```
import matplotlib.pyplot as plt
def graphique(liste_vainqueurs):
    """On représente graphiquement une liste de vainqueurs"""
    plt.close ('all')
    abscisses = [0]
    ordonnees = [0]
    for k in range(1,len(liste_vainqueurs)):
        abscisses = abscisses + [abscisses[k-1]+1]
        if liste_vainqueurs[k] == "N":
            ordonnees = ordonnees + [ordonnees[k-1]+1]
        if liste_vainqueurs[k] == "B":
```

```
            ordonnees = ordonnees + [ordonnees[k-1]-1]
    fig = plt.figure()
    plt.plot(abscisses, ordonnees)
    plt.show()
```

[103]:
```
graphique([partie_alea_vs_alea() for _ in range(500)])
```




## 1.3 Jeu entre un joueur aléatoire et une ”IA”

### 1.3.1 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

### 1.3.2 Création de l'arbre de jeu

[104]:
```
def arbre_de_jeu():
    """construit l'arbre de jeu complet d'hexapawn"""
    parties_completes = []
```

```
    plateau_initial = [["N","N","N"],[" "," "," "],["B","B","B"]]
    parties_en_cours = [[]]
    while parties_en_cours != []:
        partie = parties_en_cours[0]
        longueur = len(partie)
        couleur_a_jouer = donne_couleur(longueur)
        plateau = etat_plateau(partie)
        possibilites = coups_possibles(plateau, couleur_a_jouer)
        for coup in possibilites:
            suite_partie = partie + [coord_vers_chaine(coup)]
            plateau_suivant = etat_plateau(suite_partie)
            if fin_de_partie(plateau_suivant, couleur_a_jouer) == True:
                parties_completes = parties_completes + [suite_partie]
            else:
                parties_en_cours = parties_en_cours + [suite_partie]
        parties_en_cours.pop(0)
    return sorted(parties_completes)
```

[105]:
```
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
```

```
['B1B2', 'A3A2', 'C1C2', 'C3B2', 'A1B2', 'A2A1']
```

[105]: 134

### 1.3.3 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.

[106]:
```
def boites():
    """renvoie la matrice des états de jeux distincts et des coups␣
 ↪possibles pour l'ensemble des branches de l'arbre et toutes les␣
 ↪étapes"""
    arbre = arbre_de_jeu()
    etats = [[[etat_plateau(partie[:
 ↪j]),coups_possibles(etat_plateau(partie[:j]),donne_couleur(j)) ] for j␣
 ↪in range(len(partie))] for partie in arbre ]
    extraction = [[] for i in range(7)]
```

```
    for partie in etats:
        for j in range(len(partie)):
            situation = partie[j]
            if situation not in extraction[j]:
                extraction[j] = extraction[j] + [situation]
    longueurs = [len(extraction[j]) for j in range(7)] # pour vérifier le nombre de boites différentes
    return extraction

# variable globale contenant toutes les boites
PIOCHE = boites()

# pour activer la recherche
def recherche(plateau,tour):
    """recherche dans la liste des états possibles le plateau obtenu au tour donné"""
    global PIOCHE
    for i in range(len(PIOCHE[tour])):
        if PIOCHE[tour][i][0] == plateau:
            return i
```

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

[107]:
```
boites()[3][4][0] # pour un tour des noirs (rang 3, impaire), situation de jeu numérotée 4 dans la pioche
```

[107]:
```
[['N', 'N', ' '], ['B', 'B', 'N'], [' ', ' ', 'B']]
```

[108]:
```
recherche([['N', 'N', ' '], ['B', 'B', 'N'], [' ', ' ', 'B']],3) # retrouve le numéro de la situation qui correspond au plateau
```

[108]:
```
4
```

### 1.3.4 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`.

[ ]:

[ ]:

[109]:
```
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:
                        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`**.**

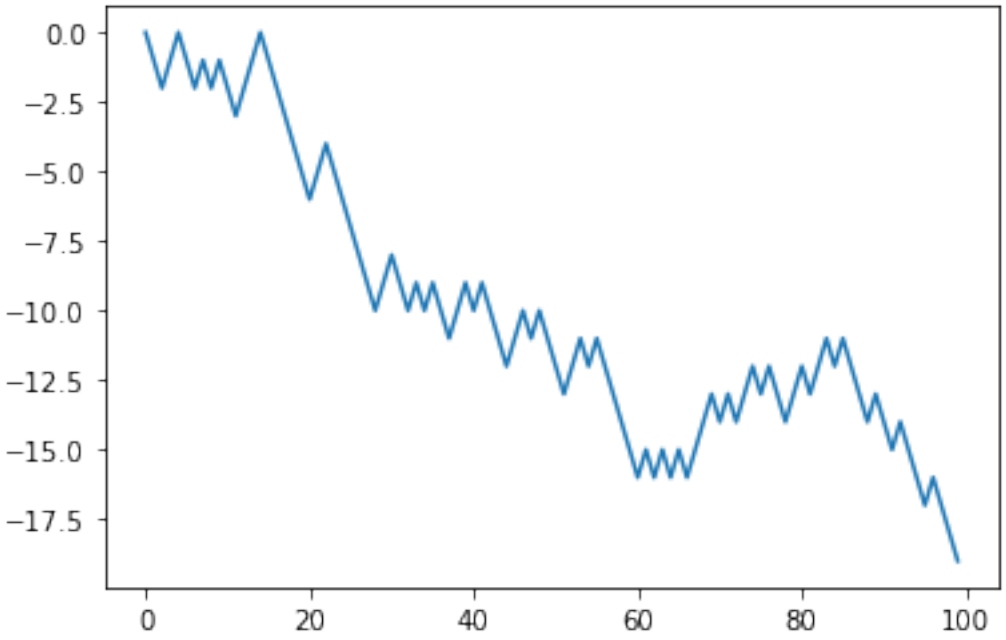
[110]:
```
print(sequence_alea_vs_ia(100))
```

```
['N', 'N', 'B', 'B', 'N', 'B', 'N', 'N', 'N', 'N', 'N', 'N', 'B', 'N', 'N', 'N',
'B', 'B', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'B', 'N', 'N', 'N', 'N', 'N',
'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'B', 'N', 'N', 'N', 'N',
'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N',
'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N',
'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N',
'B', 'N', 'N', 'N']
```

**Afficher le graphique pour la liste** `sequence_alea_vs_ia(100, punition = False, re-`

`compense = False)`. Que retrouve-t-on ?

[119]:
```
graphique(sequence_alea_vs_ia(100, punition = False, recompense = False))
```

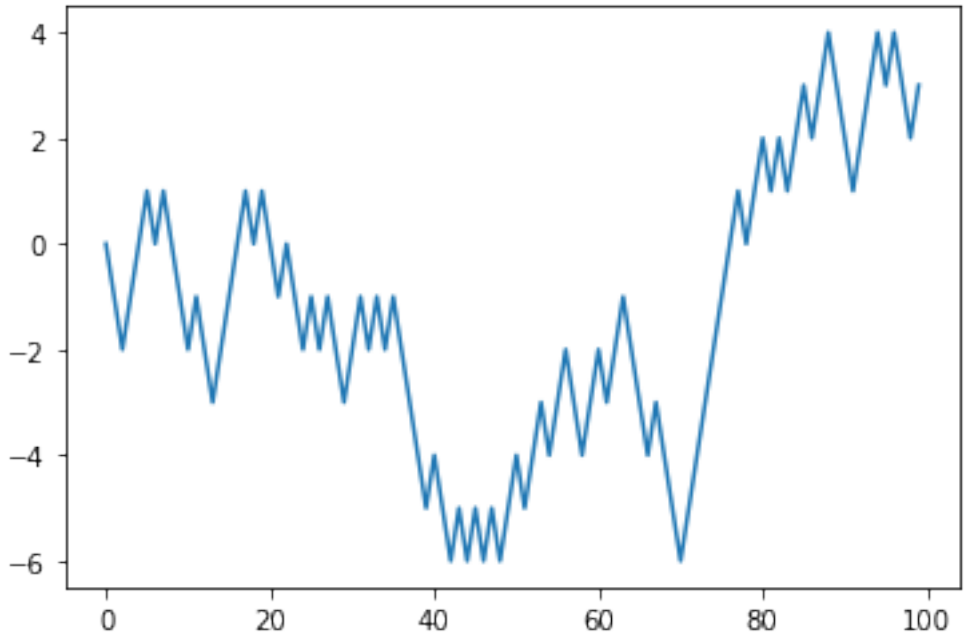



On retrouve l'évolution dans le cas d'une partie aléatoire, les blancs ayant tendance à davantage gagner (courbe qui a une tendance à la baisse)

**Afficher le graphique pour une séquence de 100 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)

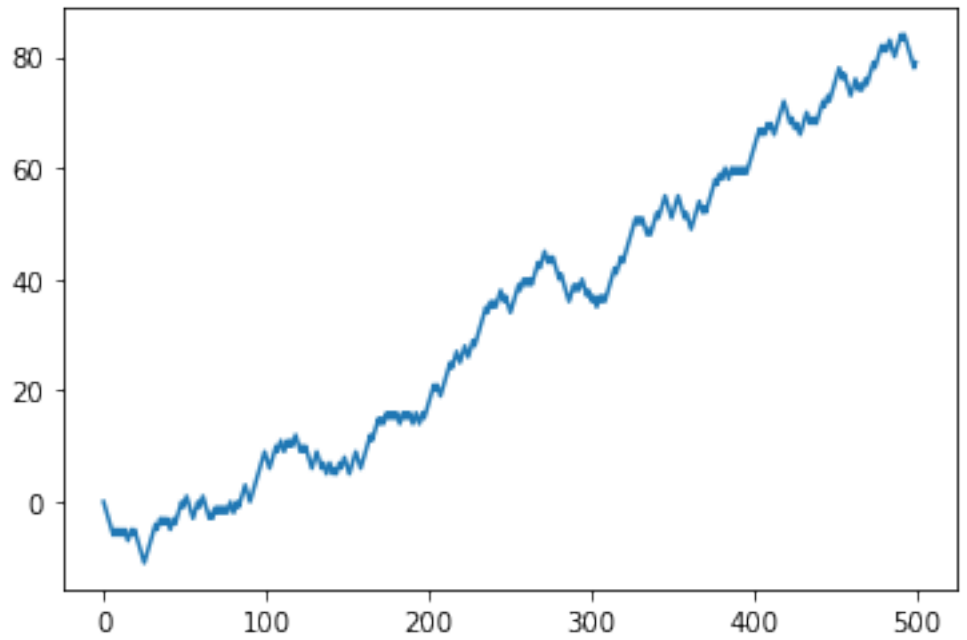
Réessayez avec un nombre de parties plus important.

[128]:
```
graphique(sequence_alea_vs_ia(100, punition = False, recompense = True))
```




L'apprentissage ne semble pas performant sur une série de 50 parties : encore beaucoup de défaites pour la machine.

[113]:
```
graphique(sequence_alea_vs_ia(500, punition = False, recompense = True))
```

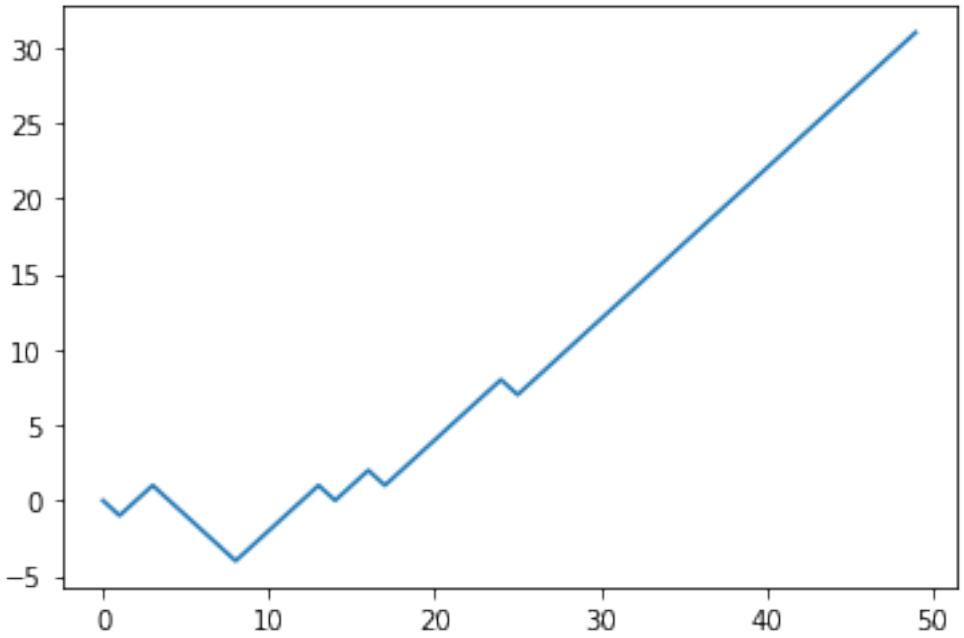


En revanche si l'on augmente le nombre de parties, la tendance est à l'augmentation de la victoire des noirs : l'apprentissage est visible.

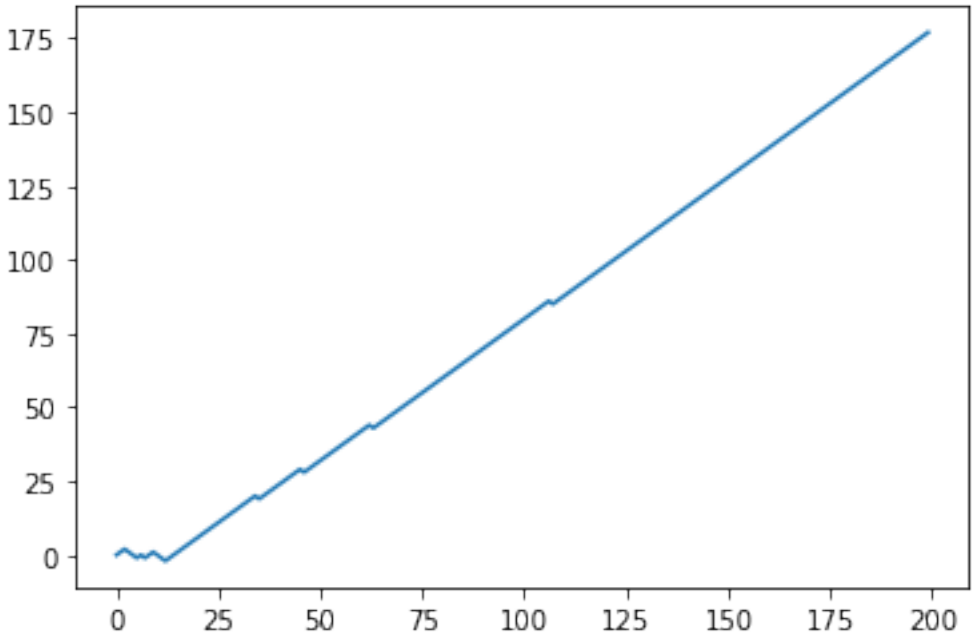
**Afficher le graphique pour une séquence de 50 parties, avec punition mais sans récompense.** Comment jugez-vous l'apprentissage de la machine ? (Afin de vous faire une idée, n'hésitez pas à ré-exécuter la cellule afin de relancer la séquence)

Tester avec 200 parties.

[114]:
```
graphique(sequence_alea_vs_ia(50, punition = True, recompense = False))
```




[115]:
```
graphique(sequence_alea_vs_ia(200, punition = True, recompense = False))
```

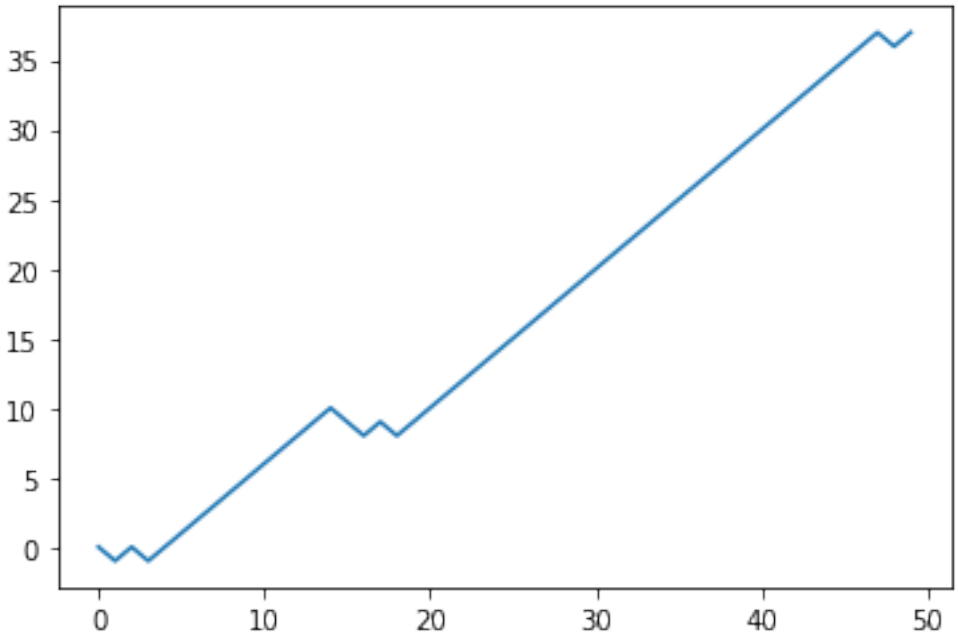



Dès 50 parties, on constate une forte tendance à la victoire pour la machine. Avec 200 parties, il n'y a plus de défaites dès la 80-100-ème partie

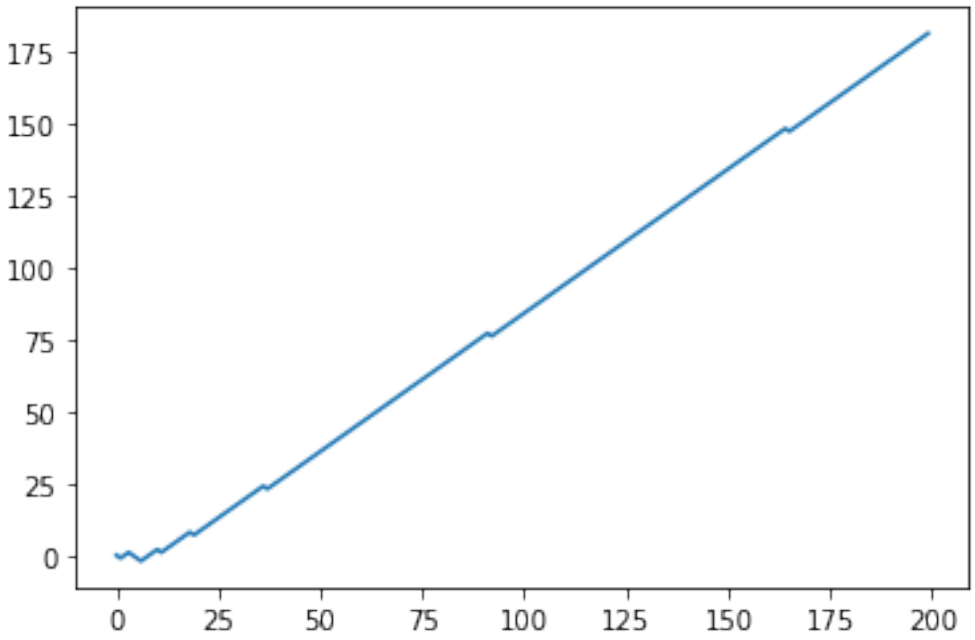
**Afficher le graphique pour une séquence de 50 parties, avec punition et avec récompense.** Comment jugez-vous l'apprentissage de la machine ? (Afin de vous faire une idée, n'hésitez pas à ré-exécuter la cellule afin de relancer la séquence)

Tester avec 200 parties.

[116]:
```
graphique(sequence_alea_vs_ia(50, punition = True, recompense = True))
```




[117]:
```
graphique(sequence_alea_vs_ia(200, punition = True, recompense = True))
```




Avec punition et récompense, on a à peu près la même courbe d'apprentissage. L'effet de la récompense ne semble pas prépondérant. Peut-être l'IA commence-t-elle à gagner plus tôt.

Ce résultat serait sûrement plus flagrant dans le cas d'un joueur humain qui jouerait des meilleurs coups et ferait apprendre l'IA plus vite. Une autre option serait de rajouter des perles à chaque état de jeu ayant mené à une victoire de la machine, afin de rendre cette partie plus probable.

[ ]:

## 1.4 Recherche de la 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.

[39]:
```
def recherche_strategie(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 = []
    liste_parties_gagnees_noirs = []
    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 = []
        liste_mouvements = []
        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
                liste_mouvements = liste_mouvements +␣
↪[coord_vers_chaine(coup)]
                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)
                    liste_mouvements = liste_mouvements +␣
↪[coord_vers_chaine(coup)]
                    if fin_de_partie(plateau, "N") == True:
                        partie_en_cours = False
                        liste_vainqueurs = liste_vainqueurs + ["N"]
                        if liste_mouvements not in␣
↪liste_parties_gagnees_noirs: # ajout de la partie gagnante
                            liste_parties_gagnees_noirs =␣
↪liste_parties_gagnees_noirs + [liste_mouvements]
```

```
                        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 sorted(liste_parties_gagnees_noirs),␣
↪len(liste_parties_gagnees_noirs)
```

[56]:
```
recherche_strategie(10000)
```

[56]:
```
([['A1A2', 'B3A2', 'B1A2', 'C3C2'],
  ['A1A2', 'B3A2', 'B1B2', 'A2A1'],
  ['A1A2', 'B3A2', 'B1B2', 'A3B2', 'C1B2', 'A2A1'],
  ['A1A2', 'B3A2', 'B1B2', 'A3B2', 'C1B2', 'C3B2'],
  ['A1A2', 'B3A2', 'B1B2', 'A3B2', 'C1C2', 'A2A1'],
  ['A1A2', 'B3A2', 'B1B2', 'A3B2', 'C1C2', 'B2B1'],
  ['A1A2', 'B3A2', 'B1B2', 'C3B2', 'C1B2', 'A2A1'],
  ['A1A2', 'B3A2', 'B1B2', 'C3B2', 'C1B2', 'A3B2'],
  ['A1A2', 'B3A2', 'B1B2', 'C3B2', 'C1C2', 'A2A1'],
  ['A1A2', 'B3A2', 'B1B2', 'C3B2', 'C1C2', 'B2B1'],
  ['A1A2', 'B3A2', 'C1C2', 'A2A1'],
  ['A1A2', 'B3A2', 'C1C2', 'A2B1'],
  ['B1B2', 'A3B2', 'A1A2', 'B2B1'],
  ['B1B2', 'A3B2', 'A1A2', 'B2C1'],
  ['B1B2', 'A3B2', 'A1A2', 'B3A2', 'C1B2', 'A2A1'],
  ['B1B2', 'A3B2', 'A1A2', 'B3A2', 'C1B2', 'C3B2'],
  ['B1B2', 'A3B2', 'A1A2', 'B3A2', 'C1C2', 'A2A1'],
  ['B1B2', 'A3B2', 'A1A2', 'B3A2', 'C1C2', 'B2B1'],
  ['B1B2', 'A3B2', 'A1B2', 'C3C2'],
  ['B1B2', 'A3B2', 'C1B2', 'C3C2', 'A1A2', 'C2C1'],
  ['B1B2', 'A3B2', 'C1C2', 'B2A1'],
  ['B1B2', 'A3B2', 'C1C2', 'B2B1'],
  ['B1B2', 'A3B2', 'C1C2', 'B3C2', 'A1A2', 'B2B1'],
  ['B1B2', 'A3B2', 'C1C2', 'B3C2', 'A1A2', 'C2C1'],
  ['B1B2', 'A3B2', 'C1C2', 'B3C2', 'A1B2', 'C2C1'],
  ['B1B2', 'A3B2', 'C1C2', 'B3C2', 'A1B2', 'C3B2'],
  ['B1B2', 'C3B2', 'A1A2', 'B2B1'],
  ['B1B2', 'C3B2', 'A1A2', 'B2C1'],
```

```
  ['B1B2', 'C3B2', 'A1A2', 'B3A2', 'C1B2', 'A2A1'],
  ['B1B2', 'C3B2', 'A1A2', 'B3A2', 'C1B2', 'A3B2'],
  ['B1B2', 'C3B2', 'A1A2', 'B3A2', 'C1C2', 'A2A1'],
  ['B1B2', 'C3B2', 'A1A2', 'B3A2', 'C1C2', 'B2B1'],
  ['B1B2', 'C3B2', 'A1B2', 'A3A2', 'C1C2', 'A2A1'],
  ['B1B2', 'C3B2', 'A1B2', 'A3B2', 'C1C2', 'B3C2'],
  ['B1B2', 'C3B2', 'C1B2', 'A3A2'],
  ['B1B2', 'C3B2', 'C1C2', 'B2A1'],
  ['B1B2', 'C3B2', 'C1C2', 'B2B1'],
  ['B1B2', 'C3B2', 'C1C2', 'B3C2', 'A1A2', 'B2B1'],
  ['B1B2', 'C3B2', 'C1C2', 'B3C2', 'A1A2', 'C2C1'],
  ['B1B2', 'C3B2', 'C1C2', 'B3C2', 'A1B2', 'A3B2'],
  ['B1B2', 'C3B2', 'C1C2', 'B3C2', 'A1B2', 'C2C1'],
  ['C1C2', 'A3A2', 'B1A2', 'B3A2'],
  ['C1C2', 'B3C2', 'A1A2', 'C2B1'],
  ['C1C2', 'B3C2', 'A1A2', 'C2C1'],
  ['C1C2', 'B3C2', 'B1B2', 'A3B2', 'A1A2', 'B2B1'],
  ['C1C2', 'B3C2', 'B1B2', 'A3B2', 'A1A2', 'C2C1'],
  ['C1C2', 'B3C2', 'B1B2', 'A3B2', 'A1B2', 'C2C1'],
  ['C1C2', 'B3C2', 'B1B2', 'A3B2', 'A1B2', 'C3B2'],
  ['C1C2', 'B3C2', 'B1B2', 'C2C1'],
  ['C1C2', 'B3C2', 'B1B2', 'C3B2', 'A1A2', 'B2B1'],
  ['C1C2', 'B3C2', 'B1B2', 'C3B2', 'A1A2', 'C2C1'],
  ['C1C2', 'B3C2', 'B1B2', 'C3B2', 'A1B2', 'A3B2'],
  ['C1C2', 'B3C2', 'B1B2', 'C3B2', 'A1B2', 'C2C1'],
  ['C1C2', 'B3C2', 'B1C2', 'A3A2']],
 54)
```

On retrouve les parties victorieuses obtenues par élagage de l'arbre au tableur

[57]:
```
recherche_strategie(10000, recompense = True)
```

[57]:
```
([['A1A2', 'B3A2', 'B1A2', 'C3C2'],
  ['A1A2', 'B3A2', 'B1B2', 'A2A1'],
  ['A1A2', 'B3A2', 'B1B2', 'C3B2', 'C1C2', 'B2B1'],
  ['A1A2', 'B3A2', 'C1C2', 'A2A1'],
  ['A1A2', 'B3A2', 'C1C2', 'A2B1'],
  ['A1A2', 'B3B2', 'C1B2', 'C3B2'],
  ['B1B2', 'A3B2', 'A1A2', 'B2B1'],
  ['B1B2', 'A3B2', 'A1A2', 'B2C1'],
  ['B1B2', 'A3B2', 'A1A2', 'B3A2', 'C1B2', 'A2A1'],
  ['B1B2', 'A3B2', 'A1A2', 'B3A2', 'C1C2', 'B2B1'],
  ['B1B2', 'A3B2', 'A1B2', 'C3C2'],
  ['B1B2', 'A3B2', 'C1B2', 'C3C2', 'A1A2', 'C2C1'],
```

```
 ['B1B2', 'A3B2', 'C1C2', 'B2A1'],
 ['B1B2', 'A3B2', 'C1C2', 'B2B1'],
 ['B1B2', 'C3B2', 'A1A2', 'B2B1'],
 ['B1B2', 'C3B2', 'A1A2', 'B2C1'],
 ['B1B2', 'C3B2', 'A1A2', 'B3A2', 'C1B2', 'A3B2'],
 ['B1B2', 'C3B2', 'A1B2', 'A3A2', 'C1C2', 'A2A1'],
 ['B1B2', 'C3B2', 'C1B2', 'A3A2'],
 ['B1B2', 'C3B2', 'C1C2', 'B2A1'],
 ['B1B2', 'C3B2', 'C1C2', 'B2B1'],
 ['B1B2', 'C3B2', 'C1C2', 'B3C2', 'A1A2', 'C2C1'],
 ['B1B2', 'C3B2', 'C1C2', 'B3C2', 'A1B2', 'C2C1'],
 ['C1C2', 'B3C2', 'A1A2', 'C2B1'],
 ['C1C2', 'B3C2', 'A1A2', 'C2C1'],
 ['C1C2', 'B3C2', 'B1B2', 'A3B2', 'A1B2', 'C3B2'],
 ['C1C2', 'B3C2', 'B1B2', 'C2C1'],
 ['C1C2', 'B3C2', 'B1B2', 'C3B2', 'A1B2', 'A3B2'],
 ['C1C2', 'B3C2', 'B1B2', 'C3B2', 'A1B2', 'C2C1'],
 ['C1C2', 'B3C2', 'B1C2', 'A3A2']],
30)
```

- Si les blancs commencent sur l'une des colonnes latérales, par exemple A1A2 (ce serait la même chose par symétrie de l'autre côté) :
  - les noirs prennent le pion qui vient d'être avancé avec leur pion central B3A2
  - si les blancs prennent le pion noir qui vient d'être joué avec leur pion central (B1A2), les noirs avancent leur pion de l'autre colonne latérale (C3C2) vers la ligne 2 pour bloquer le dernier pion blanc de la ligne 1. Les deux pions blancs restants sont bloqués. Victoire par blocage A1A2-B3A2-B1A2-C3C2 (par symétrie C1C2-B3C2-B1C2-A3A2)
  - si les blancs avancent leur pion central ou l'autre pion latéral (B1B2 ou C1C2), les noirs avancent leur pion qui était en ligne 2 dans la colonne de début de partie (A2A1) : victoire par conquête A1A2-B3A2-B1B2-A2A1 ou A1A2-B3A2-C1C2-A2A1 et les autres parties par symétrie : C1C2-B3C2-B1B2-C2C1 ou C1C2-B3C2-A1A2-C2C1
- Si les blancs commencent avec leurs pion central B1B2 :
  - les noirs prennent ce pion avec un de leurs pion latéraux, par exemple A3B2
  - si les blancs ne prennent pas ce pion, il avance et gagne par conquête (B2B1) : B1B2-A3B2-A1A2-B2B1 ou B1B2-A3B2-C1C2-B2B1, puis les parties symétriques : B1B2-C3B2-C1C2-B2B1 ou B1B2-C3B2-A1A2-B2B1
  - si les blancs prennent avec le pion de la colonne libre de noir (A1B2), les noirs avancent leur pion de la colonne opposée et gagnent par blocage (C3C2) : B1B2-A3B2-A1B2-C3C2 et la symétrique B1B2-C3B2-C1B2-A3A2
  - si les blancs prennent avec le pion de l'autre colonne (C1B2), alors le noir de cette colonne a le champ libre et gagne par conquête (C3C2-C2C1) : B1B2-A3B2-C1B2-C3C2-A1A2-C2C1 et la partie symétrique B1B2-C3B2-A1B2-A3A2-C1C2-A2A1

On voit que les stratégies gagnantes reposent sur une quinzaine de configurations différentes

[ ]: