Jeux : explorations et heuristiques

INFORMATIQUE COMMUNE - TP n° 3.8 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ☞ utiliser minimax dans le cas d'un jeu à somme nulle
- ☞ imaginer et coder des heuristiques adminissibles
- ☞ utiliser A^* pour exploration le graphe d'un jeu avec une heuristique

A Othello

Othello est un jeu de stratégie joué sur un plateau de 8x8 cases avec des pions bicolores dont une face est noire et une face est blanche. Chaque joueur se voit attribuer une couleur et doit essayer de d'obtenir le plus grand nombre de pions de sa couleur à la fin de la partie.

Le plateau possède une configuration initiale (cf. figure 1). Le jeu commence avec quatre pions placés au centre : deux noirs et deux blancs en diagonale. Le joueur noir commence la partie.

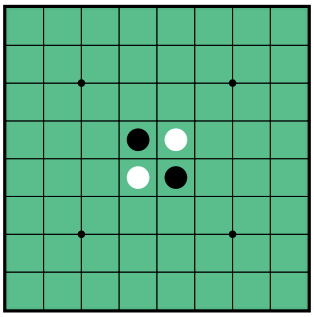

FIGURE 1 – Configuration initiale du jeu Othello

1. À chaque tour, un joueur place un pion de sa couleur **sur une case vide adjacente à un pion adverse**.
2. Pour qu'un coup soit **valide**, il est nécessaire qu'il débouche sur le retournement d'au moins un pion de l'adversaire :
   - Lorsqu'une ou plusieurs lignes de pions adverses sont encadrées entre un pion joué posé par le joueur et un autre pion de la même couleur déjà placé sur le plateau, alors on retourne tous les pions encadrées.
   - Tous les pions adverses ainsi encadrés sont retournés et changent de couleur.
3. Si un joueur ne peut jouer aucun coup valide, il **passe** son tour.

4. La partie se termine lorsque plus aucun coup n'est possible :
   - soit parce que toutes les cases sont occupées,
   - soit parce qu'aucun joueur ne peut jouer.

À la fin de la partie, le joueur ayant le plus de pions à sa couleur sur le plateau est déclaré **vainqueur**. En cas d'égalité, la partie est nulle.

B Modèle du jeu d'Othello

On chosit de modéliser le plateau du jeu par une liste de liste. Les constantes du modèle sont données ci-dessous. Les cases du plateau sont représentées par des chaînes de caractères et les joueurs par leur couleur. Pour chaque case, il est nécessaire d'étudier les directions possibles de capture des pions adverses : il y en a 8.

```
# Symboles du plateau de jeu
VIDE = "."        # case vide
BLANC = "\u25CB"  # disque blanc
NOIR = "\u25CF"   # disque noir
TAILLE = 8        # taille du plateau de jeu

NORD = (0, 1)
SUD = (0, -1)
EST = (1, 0)
OUEST = (-1, 0)
NORD_OUEST = (1, -1)
NORD_EST = (1, 1)
SUD_OUEST = (-1, -1)
SUD_EST = (-1, 1)
DIRECTIONS = [NORD, SUD, EST, OUEST, NORD_OUEST, NORD_EST, SUD_OUEST, SUD_EST]
```

Une position sur le plateau est modélisée par un couple d'entier `(ligne, colonne)`. Il s'agit maintenant de se donner des fonctions élémentaires pour appliquer les règles du jeu.

B1. Écrire une fonction de signature `init()-> list[list[str]]` qui renvoie le plateau de jeu dans sa configuration initiale

B2. Écrire une fonction de signature `afficher_plateau(plateau)` qui affiche sur la console le plateau de jeu dans sa configuration courante. Par exemple, `afficher(init())` donne le résultat indiqué sur la figure 2.

B3. Écrire une fonction de signature `sur_le_plateau(ligne: int, colonne: int)-> bool` qui renvoie `True` si la case désigné par `(ligne, colonne)` est une case du plateau de jeu, `False` sinon. On rappelle que la taille du plateau est une constante globale.

B4. Écrire une fonction de signature `valider_un_coup(plateau, ligne, colonne, joueur)-> list[tuple[int,int]]` qui renvoie la liste des pions qu'il faut retrtouner à la suite du coup joué en `(ligne, colonne)`. Par exemple, `valider_un_coup(init(),5,4,NOIR)` renvoie `[(4, 4)]`. On testera bien la capture éventuelle selon toutes les directions possibles.

B5. Écrire une fonction de signature `coups_possibles(plateau, joueur)-> list[tuple[int,int]]` qui renvoie la liste des coups possibles. Par exemple, `coups_possibles(init(), NOIR)` renvoie `[(2, 3), (3, 2), (4, 5), (5, 4)]`.

B6. Écrire une fonction de signature `copier_le_plateau(plateau)` qui renvoie une copie en profondeur du plateau de jeu.

```
  0 1 2 3 4 5 6 7
0 . . . . . . . .
1 . . . . . . . .
2 . . . . . . . .
3 . . . ○ ● . . .
4 . . . ● ○ . . .
5 . . . . . . . .
6 . . . . . . . .
7 . . . . . . . .
```

FIGURE 2 – Affichage du plateau de jeu par la fonction `affiche_plateau`

B7. Écrire une fonction de signature `jouer_un_coup(plateau, ligne, colonne, joueur)` qui renvoie un nouveau plateau représentant le plateau de jeu après le coup `(ligne, colonne)`.

B8. Écrire une fonction de signature `compter_pions(plateau, joueur)` qui renvoie le nombre de pions dont la couleur est celle de `joueur`.

B9. Écrire une fonction de signature `est_vainqueur(plateau)` qui renvoie `BLANC` si le vainqueur est le joueur en blanc, `NOIR` si le vainqueur est le joueur en noir et `None` s'il y a égalité.

C Faire jouer l'ordinateur contre lui-même

On dispose des éléments nécessaires pour faire jouer l'ordinateur selon les règles. Il nous faut maintenant implémenter :

1. la logique de contrôle du jeu, l'alternance des coups,
2. des stratégies élementaires pour choisir le prochain coup à joueur automatiquement.

Toutes les stratégies implémentées vont posséder la même signature afin de pouvoir facilement les comparer entre elles.

C10. Écrire une fonction de signature `strategie_aleatoire(plateau: list[list[str]], joueur: str)-> tuple[int,int]` qui renvoie le prochain coup à jouer si la stratégie est aléatoire, c'est-à-dire elle choisit un coup aléatoirement parmi les coups possibles. On pourra utiliser la fonction `choice` du module `random` qui choisit un élément aléatoirement dans une liste.

C11. Écrire une fonction de signature `strategie_gloutonne(plateau, joueur)-> tuple[int,int]` qui renvoie le prochain coup selon la stratégie gloutonne qui cherche à maximiser le nombre de pions retournés.

C12. En utilisant le code suivant, faire jouer l'ordinateur contre lui-même.

```
def jouer_partie(strategie_noire, strategie_blanche):
    """Faire jouer l'ordinateur contre lui-même en appliquant des stratégies"""
    plateau = init()
    joueur_courant = NOIR  # Premier joueur (règle officielle)
    impasses = 0

    afficher_plateau(plateau)

    while impasses < 2:  # un des deux joueurs peut jouer
```

```
            coups = coups_possibles(plateau, joueur_courant)

            if not coups:
                impasses += 1
                print("Le joueur ", joueur_courant, " ne peut pas jouer !")
            else:
                impasses = 0
                print("Au tour du joueur ", joueur_courant)
                if joueur_courant == NOIR:
                    coup  = strategie_noire(plateau, joueur_courant)
                else:
                    coup = strategie_blanche(plateau, joueur_courant)
                plateau = jouer_un_coup(plateau, coup[0], coup[1], joueur_courant)
                print("Le joueur ", joueur_courant, " a joué en ", coup)

            afficher_plateau(plateau)
            nb = compter_pions(plateau, BLANC)
            nn = compter_pions(plateau, NOIR)
            print("NOIRS / BLANCS -> ", nn, " / ", nb)
            joueur_courant = BLANC if joueur_courant == NOIR else NOIR

        print("Fin de partie")
        vainqueur = est_vainqueur(plateau)
        nb = compter_pions(plateau, BLANC)
        nn = compter_pions(plateau, NOIR)
        if vainqueur == BLANC:
            print("Les BLANCS ont gagnés ! ", nb, " > ", nn)
        elif vainqueur == NOIR:
            print("Les NOIRS ont gagnés ! ", nn, " > ", nb)
        else:
            print("Égalité - partie nulle !",nn,nb)
        return vainqueur

    # Programme principal
    jouer_partie(strategie_gloutonne, strategie_aleatoire)
```

D Minimax et des heuristiques

Minimax peut fonctionner sans heuristique particulière si l'arbre à explorer est de taille raisonnable. Mais, dès lors que l'arbre a explorer est grand, il est nécessaire de ne plus explorer toutes les branches et d'évaluer la position courante : c'est le rôle de l'heuristique.

Une bonne heuristique pour l'algorithme minimax calcule un score : positif pour un joueur et négatif pour l'autre. Ce score doit :

- permettre une bonne **différenciation** des positions : il doit être possible de distinguer les positions gagnantes, celles qui sont équilibrées et les perdantes,
- donner une bonne **approximation** du score réel de la position : plus cette valeur est proche de la valeur exacte obtenue par exploration complète de l'arbre, plus les décisions prises seront pertinentes,
- être **rapide à calculer** pour ne pas ralentir l'exploration de l'arbre.

Les coins, les cases jouxtant et les bordures n'ont pas la même importance. C'est pourquoi, une heuristique simple peut consister à pondérer le score des joueurs par la position de chaque pion et

fonction de la matrice suivante :

```
poids = [
    [100, -20, 10, 5, 5, 10, -20, 100],
    [-20, -50, 1, 1, 1, 1, -50, -20],
    [10, 1, 3, 2, 2, 3, 1, 10],
    [5, 1, 2, 1, 1, 2, 1, 5],
    [5, 1, 2, 1, 1, 2, 1, 5],
    [10, 1, 3, 2, 2, 3, 1, 10],
    [-20, -50, 1, 1, 1, 1, -50, -20],
    [100, -20, 10, 5, 5, 10, -20, 100]
]
```

Cette heuristique permet de bien différencier les positions non équivalente : plus le joueur a de pions sur les coins et les bordures, plus le score sera élevé. Les cases auxquelles on attribue des points positifs sont des cases qui sont difficilement retournables ou pas retournables. Celles qui ont des scores négatifs donnent accès aux cases importantes à l'autre joueur. Les pondérations sont comptés **positivement** pour le joueur et **négativement** pour l'adversaire. L'approximation du scrore est donc crédible. Enfin, le calcul est linéaire par rapport au nombre de cases du plateau, ce qui est rapide.

D13. Écrire une fonction de signature `h_simple(plateau, joueur)-> int` qui renvoie le score du plateau pour `joueur`.

D14. Écrire une fonction de signature `minimax(plateau, joueur, maximisant=True, profondeur=4) -> int, tuple[int,int]` qui renvoie le couple `score, coup` selon l'algorithme minimax et l'heuristique précédente.

On crée une fonction d'encapsulation (ou adapatateur) afin que la stratégie présente la même interface au programme de jeu :

```
def strategie_minimax(plateau, joueur, heuristique, profondeur=4):
    score, coup = minimax_ab(plateau, joueur, heuristique, maximisant=True,
        profondeur=profondeur)
    return coup

def s1(plateau, joueur):
    return strategie_minimax(plateau, joueur, h_simple, profondeur=4)

# Programme principal
jouer_partie(strategie_gloutonne, s1)
```

D15. Faire jouer l'ordinateur et comparer les différentes stratégies. Tester les stratégies aléatoire et gloutonne face à minimax.

D16. Écrire une fonction de signature `minimax_ab` améliore la fonction minimax en implémentant l'élagage alpha beta.