Fiche Jalon 02

mmc
marc-michel dot corsini at u-bordeaux dot fr

Rev. 1.a : 17 février 2022

Vous devez récupérer le fichier `jalon_02.zip` et le dézipper au même endroit que précédemment. L'objectif de ce jalon est de mettre en place différentes sortes de joueurs qui vont mettre en jeu des techniques de plus en plus complexes.
Dans cette première version, une fraction des classes à développer sont présentées.

1 Les joueurs du second jalon

Succinctement, nous allons mettre en place 4 classes de joueurs + 2 classes optionnelles - les joueurs sont **indépendants** du jeu, c'est-à-dire qu'ils vont fonctionner aussi bien sur `HexaPawn` que sur le `Morpion` que sur les jeux déjà présents dans le répertoire **Code**, les jeux contenus dans `allumettes.py` et `divide_left.py`

1. `Human` en réalité un simple menu, qui va permettre de proposer les différentes actions autorisées et qui renverra celle choisie par l'utilisateur.

2. `Randy` premier joueur numérique, qui joue au hasard une action autorisée.
3. `MinMax` qui va choisir une action autorisée pour laquelle l'algorithme prédit la plus forte évaluation.
4. `Negamax` qui fait exactement comme `MinMax` la différence réside dans l'écriture du code qui est plus concise (**bonus 1**)
5. `AlphaBeta` qui donne le même résultat que `MinMax` mais en effectuant (en moyenne) moins de calcul
6. `AlphaBetaNegamax` qui fait exactement comme `AlphaBeta` la différence réside dans l'écriture du code qui est plus concise (**bonus 2**)

2 Codage

Toutes les classes de joueurs sont dérivées de la classe abstraite `Player` qui se trouve dans le répertoire **classes**. Aucune des classes ne contient de constructeur (**pas de `__init__`**). Toutes les classes de joueur doivent définir une méthode `decision` qui reçoit en entrée un « état du jeu » et qui est dépendant du jeu auquel jouera le joueur. La méthode `decision` commence **toujours** par vérifier si c'est bien à ce joueur d'agir.

```
def decision(self, state):
    self.game.state = state
    if self.game.turn != self.who_am_i:
        print("not my turn to play")
        return None
    # a moi de jouer
```

Les autres méthodes dont on aura besoin seront nommés en les faisant précéder de __

Nous allons dans un premier temps expliciter à quoi servent les différentes méthodes des deux classes abstraites

2.1 Game

Cette classe offre plusieurs méthodes qui vont nous être nécessaires. Les jeux ayant été construits, seules les attributs et méthodes importantes sont présentées

1. `turn` permet de connaître à qui est-ce de jouer
2. `opponent` permet de connaître qui est l'adversaire
3. `winner` permet de savoir qui a gagner, `None` si pas de gagnant connu (ex-aequo, ou partie non terminée).
4. `actions` renvoie la liste (non modifiable) des actions possibles
5. `over()` renvoie `True` si la partie est terminée, `False` sinon
6. `win()` renvoie `True` s'il y a un gagnant, `False` sinon
7. `move(a)` applique l'action `a` si elle est autorisée et modifie le jeu en conséquence. **Attention** cette méthode ne renvoie rien, elle a pour effet de mettre à jour `self.game.state`

8. `undo()` permet de défaire la dernière action. Cette méthode sera utile lorsqu'on voudra « anticiper » les actions futures. **Attention** cette méthode ne renvoie rien, elle a pour effet de mettre à jour `self.game.state`

2.2 Player

Puisque les classes que nous allons écrire héritent de cette classe, il faut commencer par voir quels sont les paramètres du constructeur. Puis nous regarderons les attributs utiles, et les méthodes déjà définies, enfin la méthode à développer

2.3 Constructeur

Deux paramètres obligatoires, en premier le `nom` du joueur, c'est une simple chaine de caractères, en second le `jeu` auquel on souhaite faire jouer notre joueur numérique. Ensuite des paramètres facultatifs que nous définierons au moment où nous en aurons besoin.
Les classes dérivées n'auront pas de constructeur, elles utiliseront en conséquence le constructeur de la classe `Player`

2.4 attributs et méthodes

En lecture seule, deux attributs

1. `name` qui renvoie le nom du joueur numérique

2. `game` qui renvoie le jeu auquel on joue et qui va nous permettre d'accéder à tous les attributs et méthodes présentés en section 2.1. Imaginons que l'on veuille définir la classe `Joueur` qui hérite de `Player` et que l'on souhaite définir la méthode `nb_actions` qui renvoie le nombre d'actions possibles. On écrira :

```
class Joueur(Player):
    def nb_actions(self) -> int:
        """ renvoie le nombre d'actions possibles """
        return len(self.game.actions)
```

En lecture écriture
– `who_am_i` cet attribut ne peut prendre qu'une valeur compatible avec `self.game.turn` ou `self.game.opponent` Elle permet de définir si le joueur numérique est le joueur à qui c'est le tour de jouer (on dit, « le joueur qui a le trait ») ou son adversaire.
Les méthodes, deux sont déjà définies dans la classe `Player` et une va être spécifique à chaque classe que nous allons développer

1. `get_value(key:str)` Cette méthode va nous permettre de récupérer les paramètres supplémentaires fournis au constructeur. C'est exactement le même mécanisme que la méthode `get_parameter(key:str)` qui est présente dans la classe `Game`.
2. `estimation()` Cette méthode évalue la situation courante en fonction du joueur racine
3. `decision(state)` Cette méthode est la **seule** qu'il va falloir préciser dans chacune des classes que nous

allons développer jusqu'à la fin du semestre. Son but va être de renvoyer une action appartenant à `self.game.acti`

3 Déroulement du travail

Une fois dézipper le fichier `jalon_02.zip` vous allez trouver dans le répertoire `Code` le fichier `synopsis_02.py`. Vous allez recopier ce fichier dans `players.py`. Ne rajoutez pas de commandes **import** ailleurs que dans la partie « tests » en fin de fichier.
Rappel Les classes que vous allez écrire hérite (ou dérive) de la classe `Player`, c'est-à-dire que si vous créez la classe `Toto` il faudra mettre

```
class Toto(Player):
```

Toutes les classes doivent contenir une méthode `decision` qui commence **toutes** de la même manière

```
def decision(self, state):
    self.game.state = state # on met à jour l'état
    if self.game.turn != self.who_am_i:
        print("not my turn to play")
        return None
    # maintenant on peut travailler
```

3.1 Joueur interface humaine

La classe est `Human` On affiche le jeu, on demande à l'utilisateur de choisir une action parmi les actions autorisées

et on **renvoie** cette action.
Erreurs classiques On ne veut pas un print de l'action, on veut qu'elle soit renvoyée ! De plus si l'utilisateur choisit quelque chose qui n'est pas autorisé on recommence à lui demander l'action qu'il souhaite faire. On veut que cela fonctionne pour n'importe quel jeu qui a les bonnes propriétés, tels que ceux étudiés cette année mais aussi ceux qui sont mis à disposition dans le répertoire.

3.2 Joueur aléatoire

La classe est `Randy` la méthode de décision **renvoie** au hasard, grâce à la commande `random.choice` une des actions possibles du jeu.

3.3 Joueur MinMax récursif

La classe se nomme `MinMax`, elle va nécessiter, en plus de la méthode `decision` de deux méthodes **privées** c'est-à-dire dont le nom sera préfixé par 2 soulignés « __ »(ou « tiret du 8 », « underscore »en grand breton). Le but est de parcourir l'arbre des coups possibles jusqu'à une certaine profondeur. La profondeur sera donnée au constructeur au moment de la création

```
jeu = Hexapawn(5, 4)
moi = MinMax('toto', jeu, pf=3) # regarde 3 coups
moi.who_am_i = jeu.turn # moi joue en premier
lui = Randy('tyty', jeu)
```

```
lui.who_am_i = jeu.opponent # lui joue en second
```

La méthode `decision` va donc récupérer cette information de profondeur grâce à la commande `self.get_value('pf')`
Voici la description **algorithmique** de l'algorithme découpé en 3 parties :

```
def choix(s)
    pour chaque a_i dans ACTIONS(s) faire
         calculer s_i le nouvel etat construit par
         v_i = eval_min(s_i, pf-1)
    return a_j tel que v_j = max(v_1, .. v_k) et j

def eval_min(s, pf)
    # cherche à minimiser les gains adverses
    si s est une feuille alors retourner estimatio
    sinon
       soit s_1, .. s_k les nouveaux etats constru
       v_j = eval_max(s_j, pf -1)
       retourner min(v_1, ... v_k)

def eval_max(s, pf)
    # cherche à maximiser ses gains
    si s est une feuille alors retourner estimatio
    sinon
       soit s_1, .. s_k les nouveaux etats constru
       v_j = eval_min(s_j, pf -1)
       retourner max(v_1, ... v_k)
```

Plusieurs choses sont à comprendre. Outre l'algorithme en lui-même, il va falloir le retranscrire avec l'approche objet et les méthodes disponibles dans la classe `Game`

1. La fonction `choix` correspond à la méthode `decision`

de notre classe `MinMax`

2. Les deux autres fonctions `eval_min` et `eval_max` seront donc des méthodes cachées
3. Le paramètre `s` de la fonction `choix` correspond au paramètre `state` de la méthode `decision`
4. Le paramètre `s` des deux autres fonctions est inutile, puisque la méthode `decision` l'a stocké dans `self.game.sta`
5. Le paramètre `pf` des fonctions `eval_min` et `eval_max` est **requis** (en plus du paramètre `self`)
6. `ACTIONS(s)` s'obtient grâce à la commande `self.game.acti`
7. Le « nouvel état construit par (s, a) » est obtenu par la commande `self.game.move(a)` où `a` est un élément de `self.game.actions`
8. Lorsqu'on applique `self.game.move(a)`, il ne faut pas oublier de revenir à l'état précédent en faisant `self.game.undo()`
9. `s est une feuille` arrive dans 2 cas
   (a) soit parce que `self.game.over()` renvoie `True`
   (b) soit parce que le paramètre `pf` vaut 0
10. `estimation` est l'estimation du point de vue du joueur qui a lancé `choix`. C'est ce que calcule `self.estimation()`

3.4 Joueur Negamax récursif (optionnelle)

Cette classe **optionnelle** se nomme `Negamax` elle retourne la même information que la classe `MinMax` L'idée de cette implémentation plus concise est de s'appuyer sur la propriété

$$\forall a, b \in \mathbb{R}, \min(a, b) = -\max(-a, -b)$$

```
def choix(s)
    pour chaque a_i dans ACTIONS(s) faire
        calculer s_i le nouvel etat a partir de (
        v_i = - eval_negamax(s_i, pf-1)
    return a_j tel que v_j = max(v_1, .. v_k) et j

def eval_negamax(s, pf)
    si s est une feuille alors retourner estimatio
    sinon
       soit s_1, .. s_k les nouveaux etats constru
       v_j = - eval_negamax(s_j, pf -1)
       retourner max(v_1, ... v_k)
```

Les contraintes décriture sont identiques à celles présentées dans la section 3.3. Pour calculer l'estimation en fonction du « joueur feuille » on va utiliser la méthode `self.estimation()` et la comparaison entre `self.who_am_i` (le joueur racine) et `self.game.turn` Si les deux valeurs sont identiques **alors** l'estimation du joueur feuille est `self.estimation()`, **sinon** `- self.estimation()`

3.5 Joueur $\alpha\beta$ récursif

La classe se nomme `AlphaBeta` Il s'agit d'une optimisation (en temps) du calcul effectué par `MinMax` On va avoir 2 paramètres supplémentaires nommé `alpha` et `beta`. `alpha` désigne le score minimum possible, `beta` désigne le score maximum possible. Voici la description algorithmique de cette approche qui est une extension de la méthode de « Branch and Bound »au cas de 2 joueurs ayant des intérêts opposés.

```
def choix(s)
    pour chaque a_i dans ACTIONS(s) faire
         calculer s_i le nouvel etat a partir de (
         v_i = coupe_alpha(s_i, pf-1, alpha, beta)
    return a_j tel que v_j = max(v_1, .. v_k) et j

def coupe_alpha(s, pf, alpha, beta)
    # MIN cherche a diminuer beta
    si s est une feuille alors retourner estimatio
    sinon
       soit s_1, .. s_k les nouveaux etats constru
       i = 1
       tant que i <= k et alpha < beta faire
           v_j = coupe_beta(s_j, pf -1, alpha, bet
           si v_j <= alpha: retourner alpha
           beta = min(beta, v_j)
           i = i+1
       fait
       retourner beta

def coupe_beta(s, pf, alpha, beta)
    # MAX cherche a augmenter alpha
    si s est une feuille alors retourner estimatio
    sinon
       soit s_1, .. s_k les nouveaux etats constru
       i = 1
       tant que i <= k et alpha < beta faire
```

```
            v_j = coupe_alpha(s_j, pf -1, alpha, be
            si v_j >= beta: retourner beta
            alpha = max(alpha, v_j)
            i = i+1
        fait
        retourner alpha
```

L'analyse est la même que pour l'approche **minmax** voir section 3.3. Dans la fonction `choix` (il s'agit de la méthode `decision`) on initialise les valeurs α et β de telle sorte que l'on ait la propriété

$$\texttt{alpha} < \texttt{-self.WIN} < \texttt{self.WIN} < \texttt{beta}$$

3.6 Joueur $\alpha\beta$ negamax récursif (optionnelle)

La classe pour ce joueur est `NegAlphaBeta`, il s'agit juste de faire le lien entre l'implémentation de l'algorithme negamax pour le minmax (section 3.4) puis de l'appliquer à l'algorithme de l'alpha-béta. On n'a donc besoin que d'une fonction `coupe_alpha`, qui au lieu d'appeler `coupe_beta(s_j, pf -1, alpha, beta)` utilisera

- `coupe_alpha(s_j, pf -1, -beta, -alpha)`

Comme pour la classe `Negamax` l'évaluation au feuille, doit se faire du point de vue du joueur qui a le trait à ce niveau. Dit autrement, pour un niveau « pair » la valeur est celle de la méthode prédéfinie `self.estimation()`

pour un niveau « impair » il faudra prendre la valeur opposée.

4 Comment s'assurer du bon fonctionnement ?

Outre les tests qui vous seront fournis, il est important de tester quelques cas simples.

4.1 Tests de `decision`

La première chose à vérifier c'est que la méthode `decision` renvoie bien une action. Dans le fichier **synopsis_02.py** vous avez quelques fonctions de tests qui s'assure de cela. L'étape suivante, pour les joueurs numériques `MinMax`, `Negamax`, `AlphaBeta` et `NegAlphaBeta` c'est de vérifier qu'il vous donne la bonne réponse dans des cas simples tels que

- Il y a un coup gagnant à trouver
- Il y a un coup défense à trouver, afin d'empêcher l'adversaire de gagner au tour suivant.

Dernière étape, s'assurer que les réponses trouvées par `Negamax`, `AlphaBeta` et `NegAlphaBeta` sont identiques à celle fournie par `MinMax`

4.2 Faire des parties opposant les joueurs numériques

Dans le fichier `main_parties.py` vous avez la possibilité de confronter différentes versions de joueurs (par exemple en faisant varier la profondeur de calcul entre deux versions d'une même approche). Ce fichier contient deux fonctions :

1. `manche` qui permet de faire jouer un match entre 2 joueurs
2. `partie` qui oppose 2 joueurs sur un nombre pair de manches et qui permet de collecter des statistiques sur les résultats.