Jalon 02 : Rev. 2

marc-michel dot corsini at u-bordeaux dot fr

9 février 2021

Révision Ajout d'une information sur la solution du premier jalon. Ajout, dans la classe `Board`, d'un attribut en lecture seule `winner` permettant de savoir qui est le gagnant. Si vous gardez votre solution – uniquement dans le cas où vous avez passé avec succès les 20 tests de la méthode `win` – il est **impératif** de recopier cet attribut dans votre fichier.

Pour ce second jalon, nous allons mettre en place différents joueurs pour le jeu du « Puissance 4 » et ces variations (voir la fiche `fiche_jalon01`.
Vous allez, dans un premier temps récupérer l'archive `projet.zip`. Une fois décompressée vous obtiendrez une arborescence dont la racine est `Projet_IA`.

Chaque semaine de TD vous m'enverrez un instantanné de vos codes, et **uniquement** vos codes.
Date de l'évaluation Elle sera spécifiée sur le site.

1 Généralités

Conventions Tout au long du projet, les classes seront définies par un nom commençant par une majuscule. Les attributs et méthodes auront des identifiants anglophones, sans majuscule en première lettre, s'ils sont constitués de plusieurs mots, on utilisera le séparateur « souligné (aka tiret du 8, ou tiret bas) ».
Il n'y a aucun traitement d'erreur à moins qu'ils n'aient été explicitement demandés dans les fiches, il n'y a pas de directives *assert* dans votre code.
Les méthodes annexes devront être précédées par deux soulignés, elles ne sont pas publiques.

1.1 Héritage

Le principe de l'héritage est de permettre la factorisation de code. L'idée est de mettre un maximum en commun afin de limiter les temps de développement (écriture, recherche de bugs), et de construire des classes dérivées qui vont spécialiser certaines fonctionnalités, par exemple afin de mieux tirer parti d'une structure de données, ou en ajouter de nouvelles.

```
#!/usr/bin/env python3
#! -*- coding: utf-8 -*-

"""
Un exemple simple du principe d'heritage
"""

class A:
    def __init__(self, nom, **other):
        self.__nom = nom
        self.__args = other

    @property
    def nom(self):
        return self.__nom

    def get_key(self, key:str):
```

```
        """ renvoie la valeur d'un parametre """
        return self.__args.get(key, None)

    def __repr__(self) -> str:
        return "{0}({1}, {2})".format(self.__class__.__name__,
                                      self.nom,
                                      self.__args)

class B(A): #B herite de toutes les methodes et attributs de A

    def behavior(self):
        return "Je suis un B"


class C(A): #B herite de toutes les methodes et attributs de A

    def comportement(self):
        return "je suis un C"
```

```
mmc@hobbes-lr:$ python3 -i heritage_base.py
>>> a = A(1, truc=2, bidule=3)
>>> a
A(1, {'truc': 2, 'bidule': 3})
>>> b = B(2)
>>> b
B(2, {})
>>> c = C(3, truc=42, chose='c')
>>> c
C(3, {'truc': 42, 'chose': 'c'})
>>> a.nom
1
>>> b.nom
2
>>> c.nom
3
>>> a.get_key('truc')
2
>>> b.get_key('truc')
>>> c.get_key('chose')
'c'
>>> c.get_key('truc')
42
>>> a.behavior()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute 'behavior'
>>> b.behavior()
'Je suis un B'
>>> c.behavior()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'C' object has no attribute 'behavior'
>>> c.comportement()
'je suis un C'
>>>
```

1.2 MinMax, Negamax, $\alpha\beta$

Ces algorithmes de parcours d'arbre renvoient le même résultat, la différence entre minmax et negamax est une différence d'écriture lié à un changement de « point de vue ». La différence entre minmax et alpha-bêta est une différence de vitesse.

- L'algorithme de minmax est un algorithme de parcours explicite de l'arbre. On passe par tous les nœuds afin de déterminer la meilleure première action
- L'algoritme du negamax est un algorithme de parcours explicite de l'arbre. La simplification d'écriture est liée à un changement de point de vue pour l'évaluation des feuilles. Dans minmax, l'évaluation se fait du point de vue du sommet racine. Par exemple dans le puissance 4, si la racine est le joueur 'J', toutes les feuilles sont évaluées par rapport au point de vue de 'J'. Dans negamax, l'évaluation se fait par rapport au joueur qui doit jouer dans cette situation.
- L'algorithme $\alpha\beta$ est un algorithme de parcours implicite de l'arbre. Un sommet n'est pas évalué si l'on sait qu'il ne pourra pas améliorer le résultat déjà calculé.

2 Nouveaux fichiers

Nouveaux fichiers dans l'arborescence

1. `abstract_player` : ce fichier décrit la classe abstraite `Player`. **Toutes** les classes de joueurs dériveront de cette classe
2. `main_parties` : ce fichier propose une classe et deux fonctions.
   - `Statistics` permettant de récupérer des statistiques sur une rencontre entre deux joueurs.
   - `manche` qui prend en entrée 2 joueurs et un terrain de jeu et qui renvoie un couple de valeurs numériques
   - `partie` qui prend en entrée 2 joueurs, un terrain de jeu et un nombre entier de manches. Cette fonction renvoie une statistique résumant l'issue de la rencontre.

   Légère modification du fichier `connect4.py`. Un nouvel attribut en lecture seule a été ajouté `winner` qui renvoie le joueur gagnant.
   Par ailleurs le fichier `sol_j01.py` contient une écriture de la solution du jalon 01. Les différentes détections ont été séparées, chaque méthode respecte la syntaxe signalée
   - 4 méthodes de la forme `__alinement_XXXX` servent à traiter un alignement particulier. Le critère est défini à l'aide d'une λ-expression qui dépend de la dernière pierre jouée. Le cas « cylindrique » est traité en utilisant un modulo sur le nombre de colonnes du tablier.
     Si le nombre de pierres ainsi sélectionnées n'est pas suffisant pour un alignement, on renvoie `False` sinon, le résultat est celui de la méthode `__diagnostic`
   - La méthode `__diagnostic` prend en entrée les pierres sélectionnées triées par ordre croissant sur les colonnes, et la pierre de référence. À partir de ces informations elle crée un tableau de booléens (autant que de pierres sélectionnées) et va déterminer pour chaque pierre si elle participe ou pas à l'alignement. Pour décider, elle compare la position relative d'une pierre par rapport à la pierre de référence avec l'écart en nombre de colonnes entre la pierre et la pierre de référence sur le tablier. Si les deux valeurs concordent, la pierre est dans l'alignement, sinon elle n'y est pas et la recherche s'arrête. Dans le pire cas, la complexité est en O du nombre de pierres sélectionnées soit nbc – on ne peut pas sélectionner plus d'une pierre par colonne pour un type d'alignement particulier. La complexité en espace est du même ordre de grandeur.

2.1 Player

Cette classe est dite abstraite, car certaines méthodes ne sont pas définies, par exemple la méthode `decision`, d'autres seront retravaillées ultérieurement, par exemple `estimation`.
Attention Cette classe ne doit pas être modifiée, elle décrit le comportement général de tous les joueurs, quelque soit le jeu sous-reserve que celui-ci fournisse un certain nombre de services.

1. Le constructeur prend 2 paramètres **obligatoires** le premier `nom` est une chaîne de caractères correspondant au nom du joueur, le second `jeu` est une instance de la classe `Board`.
2. les attributs en lecture seule
   (a) `idnum` renvoie un entier unique pour chaque joueur
   (b) `name` renvoie une chaîne de caractères correspondant au nom du joueur
   (c) `game` renvoie le « Board »
3. les attributs en lecture écriture
   - `who_am_i` contient le nom du joueur dans le jeu (pour le puissance 4, ce sera donc 'J' ou 'R') cette information est importante pour les algorithmes de parcours d'arbre où l'on doit déterminer si un sommet est de type « MIN » ou « MAX ».

4. des méthodes

   (a) `__eq__` permet de comparer deux joueurs avec '=='

   (b) `clone()` permet de cloner un joueur (utile lorsqu'on veut par exemple confronter un joueur contre lui-même).

   (c) `get_value(key)` permet de récupérer un paramètre optionnel du constructeur – servira pour, la profondeur maximale dans les algorithmes de parcours d'arbre.

   (d) `decision(state)` cette méthode est le **cœur** d'un joueur automatique, elle reçoit en entrée une situation de jeu et renvoie une action **autorisée**

   (e) `estimation()` cette méthode renvoie une estimation de la situation de jeu courante. Elle a été définie le plus simplement possible. Il s'agit d'une fonction de calcul **relative** au joueur qui pose la question, i.e. celui qui prend la décision.

   ```
   # 'je' désigne la valeur de self.who_am_i
   Si 'je' suis vainqueur alors +100
   Sinon Si 'je' suis perdant alors -100
   Sinon 0
   ```

2.2 manche

Le résultat d'une manche est le nombre de coups nécessaires pour gagner la rencontre, l'ensemble des pierres est attribué au gagnant. Si la victoire a été obtenue en 10 coups par le premier joueur, le résultat est (10, 0). Si la victoire en 10 coups a été obtenue par le second joueur le résultat sera (0, 10).

2.3 partie

Fait simplement appelle à `manche` en alternant qui sera le premier joueur. Les résultats de chaque manche sont stockés et renvoyés à la fin sous la forme d'un dictionnaire (cf classe `Statistics`).

2.4 Statistics

Cette classe collecte les informations pour `partie`. Une fois stocké dans une variable, vous pourrez exploiter les informations.

```
>>> g = c4.Board(4,4,3,True) # une variation de puissance 4
>>> a = Randy('alea', g) # un joueur aleatoire
>>> s = partie(a, a, g, 4) # une partie en 4 manches contre lui-meme
>>> s
Statistics(alea_01, alea_02, Board(4, 4, 3, True))
```

- `s.reset()` réinitialise le dictionnaire
- `s.keys` renvoie les clefs principales
  1. `pv` le nombre de manches gagnées.
  2. `sigma` le nombre de coups joués, pour une manche gagnée.
  3. `avg_victories` le nombre de manches gagnées en moyenne.
  4. `avg_stones` le nombre de coups joués en moyenne, pour une manche.
- `s.subkeys` renvoie les clefs secondaires
- `s.statistics` renvoie un dictionnaire indexé sur les clefs principales et dont les valeurs sont des dictionnaires indexés sur les clefs secondaires
- `s.main_statistic(key)` renvoie l'un des 4 sous-dictionnaire en fonction de la clef
- `s.specific_statistic(subkey)` renvoie un dictionnaire avec les 4 clefs principales pour une sous-clef donnée.

```
>>> s.keys
('pv', 'sigma', 'avg_victories', 'avg_stones')
>>> s.subkeys
('alea_01', 'alea_02', 'J', 'R')
>>> s.main_statistic('sigma')
{'alea_01': 15, 'alea_02': 15, 'J': 16, 'R': 14}
>>> s.main_statistic('avg_stones')
{'alea_01': 3.75, 'alea_02': 3.75, 'J': 4.0, 'R': 3.5}
>>> s.statistics
{'pv': {'alea_01': 2, 'alea_02': 2, 'J': 2, 'R': 2},
 'sigma': {'alea_01': 15, 'alea_02': 15, 'J': 16, 'R': 14},
 'avg_victories': {'alea_01': 0.5, 'alea_02': 0.5, 'J': 0.5, 'R': 0.5},
 'avg_stones': {'alea_01': 3.75, 'alea_02': 3.75, 'J': 4.0, 'R': 3.5}}
>>> s.specific_statistic('J')
{'pv': 2, 'sigma': 16, 'avg_victories': 0.5, 'avg_stones': 4.0}
>>>
```

3 Travail pour le Jalon 02

Vous allez créer un nouveau fichier `players.py`. Vous allez ajouter le début suivant

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# this file is supposed to define all the players
from abstract_player import Player
import random
```

Vous ne travaillerez **que** dans ce fichier pour ce jalon. **Toutes** les classes que vous y créerez (une par type de joueur) seront des classes qui hériteront de `Player`, **aucune** de ces classes n'a de constructeur, **toutes** les classes auront une méthode `decision`, éventuellement d'autres qui commenceront par `__`.
Pour **toutes** les classes, la méthode `decision` commence par affecter la valeur de son paramètre `state` au jeu (voir la section 3.1 ci-dessous), puis vérifie si c'est le bon tour de jeu et ensuite renvoie une action autorisée.

3.1 Joueur aléatoire

La classe s'appelle `Randy`. Elle ne possède qu'une seule méthode

```
def decision(self, state):
```

Elle commence par affecter à l'attribut `state` de l'attribut `game` le paramètre `state`

```
self.game.state = state
```

On teste ensuite si c'est bien au joueur de jouer, si ce n'est pas le cas on affiche un message et on renvoie `None`

```
if self.game.turn != self.who_am_i:
    print("not my turn to play")
    return None
```

Ensuite elle **renvoie** au hasard grâce à la commande `random.choice` une des actions possibles du jeu

3.2 Joueur humain

La classe s'appelle `Human`. Elle ne possède qu'une seule méthode

```
def decision(self, state):
```

Après avoir affecté le paramètre `state` au jeu `self.game` et après avoir contrôlé que c'était bien le tour de jouer, on affichera l'état du damier et on fera une boucle où l'on demandera à l'utilisateur de fournir une action autorisée que l'on renverra. On réitèrera la demande jusqu'à l'obtention d'une action autorisée.

3.3 Joueur MinMax récursif

Cette classe `MinMax` possède une méthode publique

```
def decision(self, state):
```

et deux méthodes privées (préfixées par __), les deux méthodes prennent un paramètre en plus de `self` que l'on notera `pf` correspondant à la profondeur du calcul dans l'arbre. Voici la description algorithmique des 3 méthodes

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

def eval_min(s, pf)
    si s est une feuille alors retourner estimation()
    sinon
       soit s_1, .. s_k les nouveaux etats construits par (s, a_j)
       v_j = eval_max(s_j, pf -1)
       retourner min(v_1, ... v_k)

def eval_max(s, pf)
    si s est une feuille alors retourner estimation()
    sinon
       soit s_1, .. s_k les nouveaux etats construits par (s, a_j)
       v_j = eval_min(s_j, pf -1)
       retourner max(v_1, ... v_k)
```

- La première valeur `pf` utilisée dans decision est obtenue grâce à `self.get_value('pf')`
- `ACTIONS(s)` est obtenue grâce à `self.game.actions`
- Le nouvel état construit à partir de `(s, a)` est obtenu grâce à `self.game.move(a)`
- `s est une feuille` arrive dans 2 cas
  1. soit parce que `self.game.over()` renvoie `True`
  2. soit parce que le paramètre `pf` vaut 0
- `estimation` est l'estimation du point de vue du joueur qui a lancé `choix`. Ce n'est donc pas **nécessairement** celle du joueur « feuille ».

3.4 Joueur Negamax récursif

Cette classe `Negamax` possède une méthode publique

```
def decision(self, state):
```

et une méthode privée (préfixée par __), qui prend un paramètre en plus de `self` que l'on notera `pf` correspondant à la profondeur du calcul dans l'arbre.
L'idée de l'algorithme `negamax` est de s'appuyer sur le fait qu'il existe un lien entre le calcul d'un minimum et d'un maximum

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

Voici la description algorithmique des 2 méthodes

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

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

- La première valeur `pf` utilisée dans decision est obtenue grâce à `self.get_value('pf')`
- `estimation` est l'estimation du point de vue du joueur qui est `feuille` de l'arbre

3.5 Joueur $\alpha\beta$ récursif

Cette classe `AlphaBeta` possède une méthode publique

```
def decision(self, state):
```

et deux méthodes privées (préfixées par __), les deux méthodes prennent trois paramètres en plus de `self` que l'on notera `pf` correspondant à la profondeur du calcul dans l'arbre, `alpha` la borne inférieure et `beta` la borne supérieure. Voici la description algorithmique des 3 méthodes

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

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

- La première valeur `pf` utilisée dans decision est obtenue grâce à `self.get_value('pf')`

- Les valeurs initiales pour `alpha`, `beta` sont fixées de telle sorte que toute évaluation du jeu soit comprise entre ces 2 valeurs $\forall v, \alpha \leq v \leq \beta$.
- `estimation` est l'estimation du point de vue du joueur qui a lancé `choix`. Ce n'est donc pas **nécessairement** celle du joueur « feuille ».

4 Classes optionnelles

Les classes dans cette section sont uniquement pour les groupes ayant complété rapidement le jalon. L'avantage des implémentations récursives et de permettre une écriture rapide mais malheureusement peu efficace en temps et en mémoire. Le travail va donc être de réécrire les deux classes `MinMax` et `AlphaBeta` sous forme itérative. Il va donc falloir gérer une mémoire des cas non encore traités sous forme de pile, l'algorithme s'arrête lorsque la pile est vide.

4.1 Joueur $\alpha\beta$ négamax récursif

La classe pour ce joueur est `NegAlphaBeta`, il s'agit juste de faire le lien entre l'implémentation de l'algorithme négamax pour le minmax 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 fonction prédéfinie `estimation` pour un niveau « impair » il faudra prendre la valeur opposée.

4.2 Joueur MinMax itératif

La classe pour ce joueur est `MinMaxIter`

4.3 Joueur $\alpha\beta$ itératif

La classe pour ce joueur est `AlphaBetaIter`

5 Comment voir si cela marche

Très simplement, vous pouvez écrire un petit code comme celui-ci

```
import mon_projet as c4
b = c4.Board(3,3,3)
j = Human(1, b)
r = MinMax(2, b, pf=4)
j.who_am_i = b.turn
r.who_am_i = b.opponent
c = c4.Board(3,3,3)

while not c.over():
    print("Tour {}".format(c.timer))
    if c.timer %2 == 0:
        x = j.decision(c.state)
    else:
        x = r.decision(c.state)
    c.move(x)
print(c)
```

Ou bien lancer un shell à partir de `main_parties.py`

```
(base) mmc@hobbes-dev:Sandbox$ python3 -i main_parties.py
quel est le fichier de description du jeu ? mon_projet
tentative de lecture de mon_projet
>>> g = c4.Board(4,4,3,True)
```

```
>>> a = Randy('alea', g)
>>> s = partie(a, a, g, 4)
>>> s.statistics
{'pv': {'alea_01': 3, 'alea_02': 1, 'J': 3, 'R': 1},
 'sigma': {'alea_01': 20, 'alea_02': 7, 'J': 17, 'R': 10},
 'avg_victories': {'alea_01': 0.75, 'alea_02': 0.25, 'J': 0.75, 'R': 0.25},
 'avg_stones': {'alea_01': 5.0, 'alea_02': 1.75, 'J': 4.25, 'R': 2.5}}
>>> s.main_statistic('pv')
{'alea_01': 3, 'alea_02': 1, 'J': 3, 'R': 1}
>>> s.specific_statistic('R')
{'pv': 1, 'sigma': 10, 'avg_victories': 0.25, 'avg_stones': 2.5}
```