

Jalon 02

marc-michel dot corsini at u-bordeaux dot fr

3 février 2021

Attention Cette fiche est en cours de rédaction, il manque encore les parties algorithmiques pour les techniques de « Negamax, $\alpha\beta$ ». Seuls les groupes ayant validé les 20 tests de `test_win` peuvent commencer cette fiche avant le 08.02.21.

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 instantané de vos codes, et **uniquement** vos codes.

Date de l'évaluation Elle sera spécifiée sur le site.

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 Nouveaux fichiers

Deux 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.

1.1 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 fournissent 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

(b) `clone()` permet de cloner un joueur (utile lorsqu'on veut par exemple confronter deux joueurs ayant le même comportement)

(c) `get_value(key)` permet de récupérer un paramètre optionnel du constructeur – servira pour 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.

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

1.2 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).

1.3 partie

Fait simplement appelle à `manche` en alternant qui sera le premier joueur. Les résultats de chaque manche sont stockées et renvoyés à la fin sous la forme d'un dictionnaire pompeusement appelé 'Statistics'.

1.4 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 gagnée
- `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 parmi

```
>>> 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},
>>> s.specific_statistic('J')
{'pv': 2, 'sigma': 16, 'avg_victories': 0.5, 'avg_stones': 4.0}
>>>
```

2 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 ses 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 2.1 ci-dessous), puis vérifie si c'est le bon tour de jeu et ensuite renvoie une action autorisée.

2.1 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

2.2 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.

2.3 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 etat construit 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 etat construit 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

2.4 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

2.5 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

3 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.

3.1 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.

3.2 Joueur MinMax itératif

La classe pour ce joueur est `MinMaxIter`

3.3 Joueur $\alpha\beta$ itératif

La classe pour ce joueur est `AlphaBetaIter`

4 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))
    x = j.decision(c.state)
    c.move(x)
    y = r.decision(c.state)
    c.move(y)
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},
>>> 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}
```