Fiche Jalon 03

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

Rev. 1 : 22 février 2022

Dans ce troisième et dernier jalon il y a deux parties obligatoires, avec des sous-parties optionnelles, et une troisième partie optionnelle pour ceux qui veulent aller plus loin. Dans la première partie nous allons mettre en place différentes variations de la méthode $\alpha\beta$ en introduisant une mémoire pour permettre un apprentissage et le temps afin d'avoir un système efficace et agréable pour un adversaire humain. Dans la seconde partie nous allons développer des techniques, basées sur une approche de type Monte-Carlo, qui ont pour but de trouver le meilleur coup à jouer en fonction d'approches probabilistes. La dernière partie vise à combiner mémoire, temps et approches stochastiques pour mettre en place une forme d'apprentissage par renforcement.

Rappels Vous devez développer les différents joueurs un par un. Vous vérifierez que votre code est syntaxiquement correct, vous mettrez en place des petits tests simples pour vous assurez que tout semble opérationnel, et vous validerez **ensuite** vos codes avec les tests fournis. Lorsqu'une classe a été entièrement validée vous passerez à la classe suivante.
Lorsque l'on fait une opération `move(a)` il faut penser à faire une opération `undo()` pour revenir dans l'état initial du jeu.

1 Généralités

Conventions Tout au long du projet, les classes sont définies par un nom commençant par une majuscule. Les attributs et méthodes ont 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'il n'ait été explicitement demandé dans le descriptif du travail à effectuer, 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.

Pour visualiser l'impact des différentes méthodes en terme d'appels récursifs les classes dans le fichier `sol_j02.py` sont équipées de fonctionnalités supplémentaires

1. Un attribut en lecture seule `nbCalls` qui renvoie un entier correspondant au nombre d'appels récursifs d'une ou de plusieurs méthodes

2. Un décorateur `@count` placé juste avant la définition de la méthode récursive espionnée.

3. Une instruction d'initialisation dans la méthode `decision`

2 Préparation du fichier `players_mc.py`

Vous ferez une copie du fichier `synopsis_03.py`, qui contient les en-têtes nécessaires au bon fonctionnement ainsi que quelques petits tests codes. Tout le code que vous développerez pour ce dernier jalon se fera dans ce fichier.
Rappel Les classes héritent de `Player`, elles n'ont pas de constructeur, elles ont toutes une méthode `decision` dont la signature et la structure sont **identiques** à celles du jalon 02. Les classes sont à construire dans l'ordre, les classes facultatives sont indiquées au fur et à mesure.

3 Variations classiques de $\alpha\beta$

Dans cette première partie nous allons nous appuyer sur l'algorithme « alpha-beta » et son écriture « négamax ». Il va falloir commencer par recopier le code de la classe `NegAlphaBeta` qui va être modifié pour ajouter et exploiter une « mémoire ».

Pour mettre en œuvre la mémoire, préalable à tout apprentissage, nous utiliserons un dictionnaire `Python`. Les clefs, uniques, de ce dictionnaire sont données par l'attribut `hash_code` du jeu. On utilisera donc, `self.game.hash_code` dans les classes de joueurs qui stockent une information en mémoire. À chaque clef nous associerons un dictionnaire possédant 4 informations

1. `pf` : la profondeur à laquelle l'état est rencontrée, c'est un entier,
2. `exact` : un booléen qui indique si l'évaluation est exacte ce qui est le cas lorsque la partie est terminée, c'est-à-dire soit parce que on est sur une feuille avec `self.game.over()` est `True`, soit parce que l'information a été remontée depuis un état où l'évaluation était exacte.
3. `score` : la valeur de l'évaluation
4. `best_action` : l'action ayant produit le score, lorsqu'il n'y a pas d'action, parce que la partie est terminée, l'action sera `None`

Attention deux versions de la même classe sont proposées, une version « standard » et une version « heuristique ». La version heuristique est réservée à ceux qui maîtrisent le langage `Python`.

3.1 `NegAlphaBeta_Memory` (standard)

On suppose que vous avez recopié la classe `NegAlphaBeta` dans le fichier `players_mc.py`

Première étape
Renommez la classe en `NegAlphaBeta_Memory`

Seconde étape
Rattachez le dictionnaire `memory` à la méthode `decision`. Pour cela, il suffit d'ajouter **après** la fin de la méthode `decision` une instruction pour indiquer la création d'un dictionnaire vide.

```
class NegAlphaBeta_Memory(Player):
    def decision(self, state):
        ...
        return ...

    decision.memory = {}
    ...
```

Troisième étape
Avant chaque `return` dans les **deux** méthodes de la classe, il faudra enregistrer les informations dans le dictionnaire. Il faudra

1. Savoir quelle est la clef de stockage
2. Stocker le dictionnaire avec les 4 informations et les 4 valeurs

Quatrième étape
Stocker des informations c'est bien, mais ça n'a pas d'intérêt si on ne les exploite pas, c'est le rôle de cette dernière étape. Lorsque l'on atteint un nouvel état on va :

1. Récupérer la clef associée à cet état `key=self.game.hash_code`
2. Si la clef n'est pas en mémoire, on applique l'algorithme normal **sans oublier** qu'avant le `return` on stocke les informations pertinentes en mémoire.
3. Si la clef est en mémoire, 3 possibilités doivent être examinées
   (a) Le score est exact. Dans ce cas on retourne l'information mémorisée : une action si on est dans `decision`, un score si on est dans `__cut`
   (b) Le score n'est pas exact, mais la profondeur restant à explorer est inférieure ou égale à la profondeur mémorisée. Dans ce cas on retourne l'information mémorisée (une action ou un score)
   (c) Sinon il faut mettre à jour `alpha`

   ```
   alpha = max(alpha, self.decision.memory[key]['score'])
   if alpha >= beta: return beta # coupe
   ```

   Puis poursuivre comme si la clef n'avait pas été en mémoire (cas 2)

3.2 NegAlphaBeta_Memory (heuristique)

La différence avec la version « standard » réside dans l'utilisation d'une heuristique qui explore les actions en commençant par celle qui a été stockée en mémoire
`self.decision.memory[key]['best_action']`. Il n'y a aucune certitude mais en pratique cette heuristique donne de bons taux de coupes, permettant des calculs plus rapides.
Comme `self.game.actions` est un tuple `Python`, il faut le transformer en liste, enlever l'action enregistrée en mémoire de la liste et l'insérer en première position avant de faire la boucle sur les actions ainsi réorganisées. Il faudra donc utiliser consécutivement les commandes `Python` : `list()`, `index()`, `pop()` et `insert()`.

3.3 Ajout du temps

La classe `IterativeDeepening` va permettre la mise en place d'un temps limité afin de rechercher le meilleur coup. Nous allons utiliser un paramètre `secondes` qui correspondra à la durée maximale en secondes, le paramètre `pf` servira à ne pas aller au delà de la profondeur prescrite.
Nous allons utiliser la commande `time.time()` qui renvoie un nombre réel en secondes correspondant au temps écoulé depuis une valeur de référence

```
>>> time.time() ; time.localtime()
1645612263.3691223
time.struct_time(tm_year=2022, tm_mon=2, tm_mday=23, tm_hour=11, tm_min=31, tm_sec=3, tm_wday=2, tm_yday=54, tm_isdst=0)
>>> time.time() ; time.localtime()
1645612264.9691317
time.struct_time(tm_year=2022, tm_mon=2, tm_mday=23, tm_hour=11, tm_min=31, tm_sec=4, tm_wday=2, tm_yday=54, tm_isdst=0)
```

2 versions sont proposées (au choix), toutes les deux s'appuyent sur le code de la classe `NegAlphaBeta_Memory`, seule la méthode `decision` est impactée

1. Version basique

   ```
   pfmax = self.get_value('pf')
   sec = self.get_value('secondes')
   i = 0
   start = time.time()
   while i <= pfmax and time.time() - start < sec:
       calculer la meilleure action à la profondeur i
   ```

2. Version avancée, le compteur i est initialisé différemment

   ```
   Si l'etat est en mémoire: i= 1+pf en mémoire
   Sinon i = 0
   ```

4 Approches Monte-Carlo

L'approche de Monte-Carlo, dans les jeux consiste à effectuer N parties aléatoires et à regarder, statistiquement l'issue de ces N parties – bien évidemment plus N est grand, meilleure est l'approximation, mais a contrario plus long est le temps de réponse. Cette quantité (entière) sera fournie au constructeur après le mot clef `nbSim`, et sera récupérée dans les méthodes qui en ont besoin par `self.get_value('nbSim')` comme nous l'avons fait avec le paramètre de profondeur `pf`

4.1 Outils

La classe `Player` définie dans `abstract_player` fournit une méthode
`simulation(n:int=10) -> list`
qui va renvoyer une liste de 3 compteurs entiers, le premier est le nombre de victoires, le second le nombre de défaites, le dernier le nombre de matches nuls. Son rôle est d'effectuer n parties aléatoires jusqu'à la fin et d'affecter les compteurs en fonction de l'issue de chaque partie aléatoire.
Dans le fichier `synopsis_03.py` est définie une fonction
`scoring(win:int, loss:int, draw:int) -> float`
qui renvoie une valeur comprise entre -1 et 1. Cette fonction est appelée dans la suite fonction d'utilité ou, plus simplement utilité

$$\text{scoring}(w, l, d) = \frac{w - l + 0.5 \times d}{w + l + d}$$

Les différentes classes de cette section vont s'appuyer sur ces deux outils.

4.2 Monte-Carlo basique

La classe `Randy_MC` va effectuer, pour chaque action possible, `nbSim` simulations, et va renvoyer la **première** action ayant la plus grande utilité calculée par `scoring`. `Randy_MC` ne contient qu'une seule méthode

```
class Randy_MC(Player):
    def decision(self, state):
        """ get the state """
        self.game.state = state
        if self.game.turn != self.who_am_i:
            print("not my turn to play")
            return None
        nbSim = self.get_value('nbSim')
        # description algorithmique
        pour chaque action a faire
            jouer(a)
            calculer son utilité
            déjouer
        renvoyer l'action ayant la plus grande utilité
```

4.3 Monte-Carlo et AlphaBeta

Deux versions sont proposées, en fonction de votre maîtrise de `Python`. Pour les moins aguerris, vous travaillerez à modifier la classe `NegAlphaBeta`, pour les autres vous travaillerez à partir de `NegAlphaBeta_Memory` Le travail est le même, les classes seront `NegAlphaBeta_MC` et `NegAlphaBeta_Memory_MC`
Ces classes reçoivent donc deux informations `pf` la profondeur et `nbSim` le nombre de simulations à effectuer.
La modification a apporté se situe au niveau des feuilles, on distinguera le cas `self.game.over()` et le cas `pf == 0` Dans le cas où la partie est terminée, on renverra le score obtenu à partir de

`self.estimation()`, dans l'autre cas on renverra l'utilité multipliée par `self.WIN`
Remarque La modification du code de la classe de départ est minime, l'intérêt est de voir si cette petite altération de la fonction d'évaluation aux feuilles produit un joueur plus efficace.

4.4 UCB

La technique UCB « Upper Confident Bound »est une amélioration de la méthode de Monte-Carlo. Au lieu d'utiliser une répartition équitable des simulations, on va privilégier celle qui semble la plus prometteuse jusqu'à présent. Cette technique est appelée aussi technique du bandit du nom des machines à sous que l'on trouve dans les casinos. Chaque action possible est assimilée à une machine à sous, à chaque simulation on doit choisir une machine, en fonction de son historique de gains. Tel un joueur de casino on est face à ce que l'on appelle le dilemme exploitation/exploration – faut-il exploiter une machine qui semble rapporter ou bien explorer une machine qui n'a pas encore rapporté. La formule UCB à maximiser est donnée par

$$\text{utilite}_i + C\sqrt{\frac{\log(n)}{n_i}}$$

Où utilite_i est le gain moyen de la machine i ; n est le nombre total d'essais sur toutes les machines, n_i est le nombre total d'essais sur la machine i.
L'algorithme `UCB` est développé en 3 phases

```
Phase 1: initialisation
Phase 2: Gestion du dilemme
Phase 3: Choix de la meilleure action
```

La classe s'appelle `UCB`, elle reçoit en paramètre le nombre de simulations `nbSim`.

1. La phase d'initialisation consiste à effectuer, pour chaque action autorisée, une simulation – garantissant ainsi que le terme
$$\sqrt{\frac{\log(n)}{n_i}}$$
a un sens (le risque de division par 0 est écarté).

2. La phase 2, est une boucle de longueur (nbSim − 1) ∗ nAct, où `nAct` est le nombre d'actions autorisées. On effectue donc, comme dans l'approche Monte-Carlo nbSimnAct simulations. À chaque itération, on choisit l'action qui maximise la formule « UCB ».

3. La phase 3 a pour objectif de renvoyer la première action ayant la plus grande utilité.

Dans ce jalon, on choisira $C = .3$. Qui donne en général de bons résultats lorsque les calculs sont à valeur dans $[-1, 1]$

5 Pour aller plus loin

Cette dernière partie n'est à considérer que si toutes les étapes intermédiaires ont été réalisées avec succès.

5.1 `IterativeDeepening` et Monte-Carlo

Première amélioration assez simple, mixer l'approche de « l'approfondissement itératif » avec les simulations. La classe se nomme `IterativeDeepening_MC` le travail est similaire à ce qui a été fait lorsque l'on a combiné « alpha-beta » et approche de Monte-Carlo, c'est-à-dire que l'on va scinder l'évaluation aux feuilles en fonction de la fin de partie ou non.
Pour pouvoir fonctionner cette classe aura besoin de 3 informations

1. `pf` : la profondeur maximale à examiner
2. `secondes` : le nombre de secondes pour faire les calculs
3. `nbSim` : le nombre de simulations au niveau des feuilles

5.2 UCT

L'approche UCT pour « UCB Tree »vise à étendre le principe de simulations UCB dans un arbre de jeu. L'idée consiste à choisir les actions en suivant la politique UCB jusqu'à atteindre une feuille. Puis à développer aléatoirement l'arbre à partir de cette feuille, et enfin à remonter le résultat (victoire, défaite ou nul) jusqu'à la racine. Pour pouvoir fonctionner, il est nécessaire d'utiliser une mémoire (sous forme de table de transpositions) afin de garder d'une fois sur l'autre les informations. La structure de l'algorithme est similaire à celle d'une approche « négamax », sauf qu'au lieu d'explorer chaque « fils » d'un sommet, on va choisir le fils développé en suivant une politique « UCB ».

5.3 Besoins et algorithmes

On va avoir besoin d'une mémoire `self.decision.memory` dont les clefs sont obtenues par `self.game.hash_code`. L'information est stockée sous la forme d'une structure qui permet d'associer à chaque action un triplet de compteurs (victoire, défaite, nul). Le constructeur de la classe accepte 2 paramètres

- `pf` la profondeur à explorer, si elle n'est pas fournie, on explorera à la profondeur 1 – les résultats attendus sont alors similaires à ceux obtenus par la classe `UCB`.
- `nbSim` le nombre de simulations, si la valeur n'est pas fournie, on ne fera qu'une seule simulation.

On va avoir une méthode qui va renvoyer l'action maximisant la politique « ucb », une méthode qui va parcourir l'arbre en profondeur suivant l'approche « négamax » sauf qu'au lieu de renvoyer une valeur numérique dont il suffisait de changer le signe, on va renvoyer 3 compteurs et il va falloir intervertir le compteur des victoires avec le compteur des défaites puisque les points de vue sont opposés.

5.3.1 `decision`

La méthode principale est immédiate et est fournie ci-dessous

```
def decision(self, state):
    self.game.state = state
    if self.game.turn != self.who_am_i:
        print("not my turn to play")
        return None
    pf = self.get_value('pf')
    nbSim = self.get_value('nbSim')
    pf = 1 if pf is None else pf
    nbSim = 1 if nbSim is None else nbSim
    self.__simulations(pf, nbSim) # descente dans l'arbre
    return self.__ucb_policy() # renvoie l'action
```

5.3.2 ucb_policy

Un peu plus délicat à mettre en action

```
def __ucb_policy(self):
    """ renvoie l'action maximisant
        utilite(a_i) + 0.3 * math.sqrt( math.log(n) / n_i)
        a_i = la ième action
        n = sum_i n_i
        n_i = nombre de simulations faites en commençant par a_i
    """
    return a_best # meilleure a_i
```

Les informations sont dans `self.decision.memory` dont les clefs sont fournies par `self.game.hash_code`, la valeur est une structure permettant de retrouver, pour chaque action de `self.game.actions` un triplet de compteurs.
Par exemple, si pour un certain état s, il y a 4 actions possibles on doit avoir dans `self.decision.memory[s]` la valeur

```
{a1: [v1, d1, n1], a2: [v2,d2,n2], ..., a4:[v4,d4,n4]}
```

Ou encore

```
[[v1, d1, n1], [v2,d2,n2], ..., [v4,d4,n4]]
```

L'utilité de l'action `a_i` est obtenue par `scoring(vi, di, ni)`, la quantité `n_i` est simplement `vi+di+ni`

5.3.3 simulations

La partie la plus complexe de l'algorithme

```
def __simulations(pf, nbSim):
    """ keep local creation, get next level counters
        invert win/loss when updating counters
    """
```

```
    get the old scores for each action
    _sum = [0,0,0]
    if pf == 0; return _sum

    if self.game.over():
       get counters for None
       increase counters for None by nbSim
       increase _sum by nbSim
       return _sum

    for a in self.game.actions:
        if a has counters > 0: continue # no new simulation
        make nbSim simulations starting by a until game.over()

        increase counters for a
        increase _sum
    # all actions have at least one simulation
    # now performs the best action and get result
    next_action = self.__ucb_policy()
    self.game.move(next_action)
    _result = self.__simulations(pf-1, nbSim)
    self.game.undo()
    update counters for next_action with _result # win <-> loss
    update _sum with _result # win <-> loss

    return _sum
```

La variable `_sum` comptabilise toutes les simulations à ce nœud de l'arbre. Les compteurs qui remontent doivent intervertir les valeurs des compteurs de victoires et de défaites puisqu'ils proviennent d'un joueur antagoniste. Les compteurs sont indexés par les actions, quand on a une fin de partie, il n'y a pas d'action ultérieure d'où le choix `None`.

5.4 Fonction d'évaluation

Le but va être d'améliorer la performance de tous les joueurs numériques en modifiant la fonction d'évaluation. Malheureusement cette fonction d'évaluation est dépendante du jeu. Il va falloir choisir la classe du joueur que l'on souhaite spécialier, et choisir le jeu (Morpion ou Hexapawn) pour lequel la fonction d'évaluation va être altérée. La nouvelle classe s'appellera `Ultimate`
À titre d'exemple, voici la version de la classe basée sur la classe `NegAlphaBeta_Memory` spécialisée pour le jeu des allumettes.

```
class Ultimate(NegAlphaBeta_Memory):
    """ spécialisation pour le jeu des allumettes """
    def decision(self, s): return super().decision(s)
    decision.memory = {}
```

```
def estimation(self):
    if self.game.__class__.__name__ == 'Matches' and not self.game.over():
        _prendre = self.game.get_parameter("prendre")
        _board = self.game.board[0]
        _who = 1 if self.game.turn == self.who_am_i else -1
        if _prendre:
            if _board%4 == 0: return _who* -self.WIN
            else: return 0
        else:
            if _board%4 == 1: return _who*-self.WIN
            return 0

    return super().estimation()
```

1. Comme on souhaite avoir une mémoire spécifique, on est obligé de redéfinir la méthode `decision` qui ne fait rien d'autre qu'utiliser celle de la classe parente, une fois fait on peut définir `decision.memory` et l'initialiser.

2. On spécifie ensuite la méthode d'évaluation. Comme on veut que la classe marche au moins aussi bien que la classe parente pour les jeux, on va dans un premier temps détecter la nature du jeu et si ce n'est pas notre spécialité, on renvoie le résultat par défaut.

3. Dane le jeu des allumettes, une analyse assez simple permet de trouver que si on a pour objectif de prendre la dernière allumettes, avoir un nombre d'allumettes multiple de 4 est une situation perdante. Au contraire, quand l'objectif est de ne pas prendre la dernière, on est dans une situation perdante si le nombre est $m = 4p + 1$.

4. On veut détecter la nature du jeu, il faut donc récupérer le nom de la classe qui est une chaîne de caractères, de plus on ne souhaite pas interférer lorsque la partie est terminée, puisque la fonction d'estimation par défaut fait très bien le travail.

Ne reste plus qu'à vérifier l'efficacité pour le jeu des allumettes en utilisant les méthodes définies dans `main_parties`. Si la fonction définie est efficace, elle doit favoriser la victoire de la classe `Ultimate` face à son adversaire. De même on va vérifier que les deux classes `NegAlphaBeta_Memory` et `Ultimate` ont le même comportement pour un jeu non optimisé – c'est-à-dire que l'on attend un match nul à l'issue de la rencontre.

5.4.1 Jeu optimisé

```
>>> g = jeu.Matches(13, True)
>>> a = Ultimate('a', g, pf=1)
>>> b = NegAlphaBeta_Memory('b', g, pf=1)
>>> Ultimate.decision.memory = {}
>>> NegAlphaBeta_Memory.decision.memory = {}
>>> s = partie(a,b, g, 4)
>>> s.statistics
{'pv': {'a_01': 8, 'b_02': 0, 0: 4, 1: 4}, 'moves': {'a_01': 60, 'b_02': 0, 0: 28, 1:
```

```
>>> g = jeu.Matches(13, False)
>>> a = Ultimate('a', g, pf=1)
>>> b = NegAlphaBeta_Memory('b', g, pf=1)
>>> Ultimate.decision.memory = {}
>>> NegAlphaBeta_Memory.decision.memory = {}
>>> s = partie(a,b, g, 4)
>>> s.statistics
{'pv': {'a_03': 8, 'b_04': 0, 0: 4, 1: 4}, 'moves': {'a_03': 60, 'b_04': 0, 0: 32, 1:
```

5.4.2 Jeu non optimisé

```
>>> g = jeu.Divide(13, 17)
>>> a = Ultimate('a', g, pf=1)
>>> b = NegAlphaBeta_Memory('b', g, pf=1)
>>> s = partie(a, b, g, 4)
>>> s.statistics
{'pv': {'a_01': 4, 'b_02': 4, 0: 0, 1: 8}, 'moves': {'a_01': 64, 'b_02': 64, 0: 0, 1:
```