Aide jalon 02

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

Rev. 4 : 22 Fevrier 2022

Vous avez deux exercices à faire pour le rendu du 11 mars

1 Créer le fichier `players.py`

Je ne pensais pas que ce soit un problème. J'ai déjà signalé que votre zone de jeu était le répertoire `Code` vous ne **devez** pas aller ailleurs pour travailler, ce qui évitera pas mal de « ça ne marche pas ». Pour recopier un fichier, il y a plein de manière de procéder

- Depuis un terminal, vous utilisez la commande `copy` (sous windows) ou `cp` sous Mac ou Linux
- Avec la souris, vous utilisez le menu contextuel « copy »
- Avec votre outil de programmation (idle, notepad++, spyder3, vscode, ...). Vous ouvrez le fichier est vous utiliser la commande « enregistrer sous » ou « save as »

Bref, ...

2 Généralités

On vous demande de créer des joueurs numériques, afin de pouvoir jouer à un jeu. Ces joueurs numériques sont des classes qui vont dériver (hériter) d'une classe déjà construite `Player` permettant de garantir que toutes les fonctionnalités nécessaires soient présentes.
Pour indiquer qu'une classe dérive `B` d'une autre classe `A`, on utilise, en **python** la syntaxe

```
class B(A):
```

L'intérêt de ce processus d'héritage, est de minimiser le travail du programmeur. Il va se contenter de spécifier les « comportements » (c'est-à-dire les méthodes ou attributs) différents ou supplémentaires de la classe `B` par rapport à sa classe parente `A`.

Pour les différents types de joueurs, il va falloir définir la méthode `decision` dont le but est de renvoyer l'action que souhaite faire le joueur en fonction de l'état du jeu.

- `self` à l'intérieur de la classe va permettre d'accéder à toutes les informations disponibles pour un joueur
- `self.game` à l'intérieur de la classe va permettre d'accéder à toutes les informations disponibles sur le jeu auquel le joueur joue.

On vous demande de démarrer votre code de la méthode `decision` par la séquence d'instructions :

```
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
```

la méthode `decision` reçoit une information extérieure (le paramètre `state`). La première instruction stocke la valeur dans `self.game.state` c'est à dire dans l'attribut `state`, de l'attribut `game` du joueur. Dit autrement, le joueur dispose de sa propre version du jeu, sur laquelle il va pouvoir travailler.
La seconde instruction compare la valeur de `self.game.turn` (qui correspond à savoir à qui est le tour de jeu) avec la valeur de `self.who_am_i` (qui correspond au rôle du joueur) – dit autrement `self.game.turn` dit « c'est à 'X' »de jouer, tandis que `self.who_am_i` dit « je suis le joueur 'X' (ou 'O') ». Si les deux valeurs ne correspondent pas, le joueur numérique renvoie `None` indiquant qu'il n'a pas à prendre de décision puisque ce n'est pas son tour de jeu.

Si les deux valeurs `self.game.turn` et `self.who_am_i` sont identiques il faut renvoyer l'action que le joueur numérique souhaite faire.
Les actions d'un jeu, sont contenues dans l'attribut `actions` du jeu, et la méthode `decision` de **tous les joueurs numériques** doit renvoyer un élément appartenant à cet attribut.

3 Rappels

Quelques points importants

1. On ne crée pas une nouvelle classe tant que la première n'est pas complètement faite et validée par les tests fournis. L'ordre des classes est voulu, pour vous permettre d'aller du simple au complexe, vouloir faire `MinMax` alors que vous n'avez pas réussi à faire `Human` c'est comme vouloir faire le marathon de New York alors que vous n'avez pas le droit de quitter le territoire national.
2. Prenez l'habitude de tester vos codes, pas uniquement avec les testcodes fournis ou les tests de `main_tests` mais directement dans le shell. Pour cela vous pouvez vous inspirer des testcodes, les étapes sont simples :
    - importer la classe du jeu `from ... import ....`
    - creer le jeu

- creer le joueur en lui donnant une chaine de caractères ou un entier qui correspondra à son nom, et le jeu
- indiquer si le joueur est le premier ou le second joueur

```
>>> from hexapawn import Hexapawn
>>> jeu = Hexapawn(4,4) # version simple
>>> moi = Human('tyty', jeu) # un joueur interface pour hexapawn
>>> moi.who_am_i = jeu.opponent # je joue en second, jeu.turn pour commencer
```

Vous pouvez alors afficher le jeu directement ou afficher le jeu du joueur, de même que vous pouvez consulter les actions possibles ou l'état du jeu ou si la partie est terminée

```
>>> print(jeu)
>>> print(jeu.actions)
>>> print(jeu.state)
>>> print(jeu.over())
>>> print(moi.game)
>>> print(moi.game.actions)
>>> print(moi.game.state)
>>> print(moi.game.over())
```

3. Les noms des classes sont obligés, Human et pas `Humain`, Randy et pas `Random`, etc

4 Human

Pour cette classe la difficulté réside dans le fait qu'il faut qu'elle soit fonctionnelle quelque soit le jeu. C'est-à-dire que l'on ne veut pas être dépendant de la syntaxe des actions.

1. Pour le jeu des allumettes classe Matches, les actions sont des chiffres.
2. Pour le jeu des boîtes classe Divide, les actions sont des paires constituées d'une lettre et d'un chiffre.
3. Pour le jeu hexapawn classe Hexapawn, les actions sont des paires de couple, chaque couple étant un numéro de ligne et un numéro de colonne
4. Pour le jeu du morpion classe Morpion, les actions sont des couples constitué d'un numéro de ligne et d'un numéro de colonne.

De plus, on veut être certain que l'action retournée est valide c'est-à-dire qu'il s'agit bien d'une des actions autorisées par le jeu. Hors, on sait que les actions autorisées appartiennent à `actions` qui est un attribut des objets de la classe `Game`.

4.1 Où trouver les actions autorisées ?

Très simplement, elles sont dans `self.game.actions`. Cet attribut est un tuple (qui est utilisable comme une liste grâce aux crochets. Il suffit de ne manipuler les actions qu'en fonction de leur position dans cette structure. Par exemple, pour dire que l'on souhaite la première des actions possibles on écrira

```
self.game.actions[0]
```

Tout comme les listes **python**, on peut connaître la taille d'un tuple grâce à la commande `len`

4.2 Que faut-il faire ?

Il suffit de demander à l'utilisateur de choisir le numéro de l'action à effectuer en utilisant la commande `input`. Cette commande prend en paramètre un message qui sera afficher à l'écran pour l'utilisateur, **et renvoie** une chaîne de caractères. Il faudra s'assurer que la réponse de l'utilisateur est transformable en un entier.
Les chaînes de caractères disposent de nombreuses méthodes, pour les connaître on tape dans un shell python la commande `help(str)`[1], **python** utilise des commandes de la forme `isXXX()` permettant de savoir (réponse `True` ou `False`) si une entité possède la caractéristique `XXX`

L'algorithme est

```
Afficher pour chaque action l'entier que doit saisir l'utilisateur
rep = ''
Tant que rep n'est pas un nombre ou que int(rep) n'est pas un numero autorise
     rep = input("Quel est votre choix ? ")
trouver l'action a partir de son numero
renvoyer action
```

Dans un shell on peut tester des exemples simples ...

```
>>> actions = [ 'a_1', 2, 3, ((0,1),(0,2)), (2,3), ('A', 3) ]
>>> rep = input("Quel est votre choix ? ")
Quel est votre choix ? a
>>> rep
'a'
>>> rep = input("Quel est votre choix ? ")
Quel est votre choix ? 1
>>> rep
'1'
>>> rep.isnumeric()
True
>>> rep.isdigit()
True
>>> rep.isdecimal()
```

[1] `help` s'applique à n'importe quelle classe. Par exemple `help(Hexapawn)`

```
True
>>> rep = '1a'
>>> rep.isdigit()
False
>>> rep = '15'
>>> rep.isdigit()
True
>>> rep = int(rep)
>>> rep
15
>>> actions
['a_1', 2, 3, ((0, 1), (0, 2)), (2, 3), ('A', 3)]
>>> actions[rep]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> actions[0]
'a_1'
>>> actions[3]
((0, 1), (0, 2))
>>> actions[5]
('A', 3)
```

4.3 Mise en œuvre

Il s'agit simplement de faire une boucle jusqu'à obtenir une information qui soit, à la fois, un nombre et qui soit un index compatible avec la liste des actions. Pour cela une solution simple est d'avoir une variable booléenne qui passe à **True** lorsque toutes les conditions sont vérifiées

```
isOK = False
while not isOK:
    rep = input("Entrez votre choix ")
    if rep.isnumeric():
        idx = int(rep) # on sait qu'on peut faire la transformation
        if idx >= 0 and idx < len(self.game.actions):
            isOK = True # on est dans le bon intervalle
            action = self.game.actions[idx] # on a l'action
return action
```

5 Randy

L'objectif est ici de choisir aléatoirement une action parmi les actions autorisées. En début du fichier on a fait `import random`, il suffit d'utiliser une méthode de random pour effectuer ce choix. Faites `help(random)` dans le shell pour choisir la méthode la plus appropriée.

5.1 Mise en œuvre

Après avoir fait `help(random)` on trouve 3 instructions potentiellement intéressante

1. `choice(self, seq)` « Choose a random element from a non-empty sequence »
2. `randint(self, a, b)` « Return random integer in range [a, b], including both end points »
3. `randrange(self, start, stop=None, step=1)` « Choose a random item from range(start, stop[, step]) »

On peut faire `random.choice(self.game.actions)` ou encore

```
idx = random.randint(0, len(self.game.actions) -1)
self.game.actions[idx]
```

Ou encore

```
idx = random.randrange(len(self.game.actions))
self.game.actions[idx]
```

6 MinMax

Un algorithme n'est pas un programme, c'est une description *abstraite* d'un processus. vous devez faire un travail de transformation entre l'algorithme et l'implémentation.
Un algorithme ne dépend pas d'un style de programmation, il y a un travail de compréhension pour passer d'une description algorithmique à sa réalisation.
L'algorithme fait état de 3 fonctions `choix`, `eval_min`, `eval_max`. Il faut d'abord s'occuper de leurs rôles et de leurs signatures

- `choix` cherche une action, c'est la fonction principale. Elle appelle `eval_min`
- `eval_min` cherche une valeur numérique minimale, elle appelle `eval_max`
- `eval_max` cherche une valeur numérique maximale, elle appelle `eval_min`

Ces fonctions servent à parcourir un arbre en profondeur d'abord (se reporter au cours sur min-max). Mais, au lieu de parcourir un arbre statique, on va le construire dynamiquement ce qui permet de minimiser l'espace mémoire occupé. Dans un arbre statique, tout l'arbre est disponible en mémoire, dans une construction dynamique, seule la branche[2] sur laquelle on travaille est présente en mémoire.
`choix` s'applique à la racine de l'arbre, `eval_min` s'applique aux sommets gérés par l'adversaire du joueur « racine ». Tandis que `eval_max` s'applique aux sommets gérés par le joueur « racine ».
L'algorithme de `choix` cherche l'action qui correspond à la meilleure évaluation renvoyée par la fonction `eval_min`. Pour trouver cette action, on peut soit stocker ces valeurs, puis chercher l'optimum (algorithme 1), ou bien on peut garder à tout moment la meilleure action et la meilleure valeur associée (algorithme 2). Cette seconde version est moins couteuse en mémoire.

[2] une branche est une chaîne joignant la racine à un sommet de l'arbre.

```
def choix(s):
    valeurs structure de stockage des valeurs
    actions_vues structure de stockage des actions evaluees
    pour chaque action a de s faire
         stocker a dans actions_vues
         calculer s' à partir de s et a
         stocker eval_min(s', pf-1) dans valeurs
    trouver a_best dans actions_vues en fonction de valeurs
    retourner a_best
```

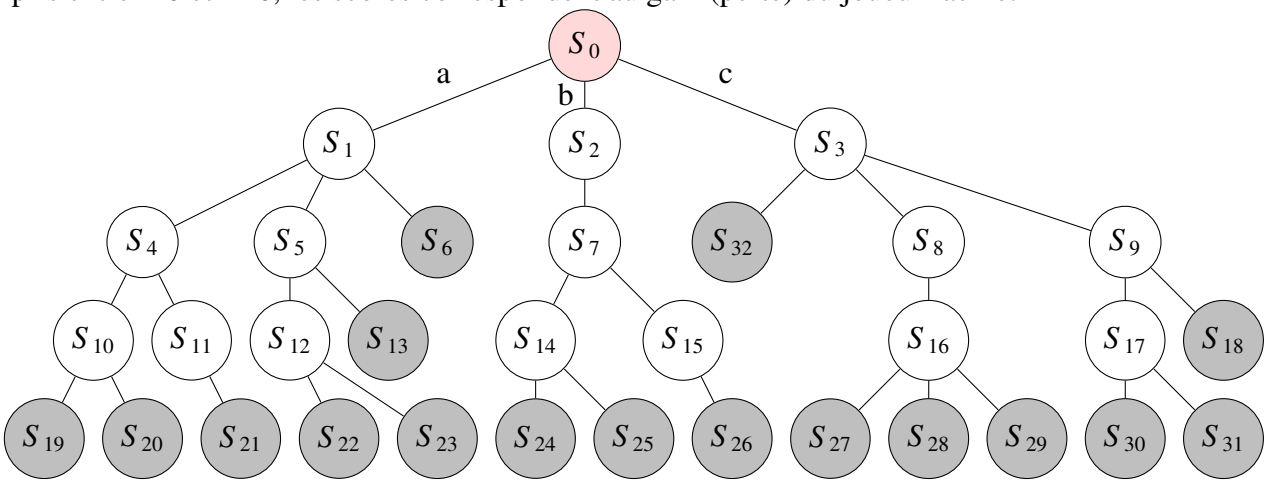
FIGURE 1 : Algo : stocker et chercher

```
def choix(s):
    v_best = - infini # -1000 suffira ici
    a_best = None
    pour chaque action a de s faire
         calculer s' à partir de s et a
         v = eval_min(s', pf-1)
         si v > v_best
            v_best = v
            a_best = a
    retourner a_best
```

FIGURE 2 : Algo : garder à jour

6.1 Exercice de résolution

Cet exercice vise à vérifier que vous avez bien compris le fonctionnement de l'algortihme du minmax pour lequel on vous donne l'arbre complet. La fonction d'évaluation autorise des scores compris entre -20 et +20, les scores correspondent au gain (perte) du joueur racine.



Les évaluations aux feuilles sont décrites dans la table suivante

sommets	6	13	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
score	1	−5	5	19	8	−10	5	2	20	5	−20	2	10	13	4	8	1

Exercice 1
Rappel Les feuilles de l'arbre sont les sommets qui ne sont pas développés (soit parce qu'il y a eu détection de fin de partie, soit parce que la profondeur maximale (4) a été atteinte). La racine est le sommet où se place la décision.

1. Quelle est la décision à la racine ?
2. Donnez les valeurs des 3 sommets successeurs de la racine qui justifie cette décision.
3. Donnez l'ordre de visite des sommets de l'arbre, l'ordre est obtenu par le premier accès au cours du parcours en profondeur d'abord.
4. Y-a-t-il des sommets non visités par l'algorithme du minmax ?
5. Donnez la valeur finale renvoyée par les sommets de l'arbre sauf celles des feuilles et celle des 4 premiers sommets de l'arbre S_0, S_1, S_2, S_3.

6.2 Réalisation

On impose une méthode `decision` dont le but est de trouver l'action à réaliser. Elle correspond à la fonction `choix`. On a besoin de deux fonctions annexes, qui doivent être privées il suffit de leur donner un nom qui commence par `__`. Nos trois méthodes sont `decision`, `__eval_min` et `__eval_max`.

- Les paramètres de `decision` sont imposés, il s'agit de `self` et `state`. Puisque on doit faire en première instruction (voir fiche) `self.game.state = state` l'état du jeu est accessible par l'attribut `self.game.state`. Lorsque l'on utilise `self.game.move(a)` cela

modifie l'état du jeu. L'opération inverse est `self.game.undo()`. Les actions autorisées sont contenues dans `self.game.actions`.
La profondeur est fournie dans le constructeur, pour y accéder il faut faire `self.get_value('pf')`. Et, on doit le faire une seule fois.

```
def decision(self, state):
    self.game.state = state # mise à jour de l'état initial
    # traitement bon joueur oui/non
    if self.game.turn != self.who_am_i:
        print("not my turn to play")
        return None
    # récupération du paramètre
    pf = self.get_value('pf')
    # recherche de la meilleure action
    # < ici on fait la mise en oeuvre de l'algorithme choix >
    return meilleure_action
```

- **Attention** Dans un un jeu `move(a)` ne renvoie rien, par contre elle modifie l'attribut `state` du jeu

```
>>> from allumettes import Matches
>>> jeu = Matches(17)
>>> print(jeu.state)
(17, 0)
>>> print(jeu.move(2))
None
>>> print(jeu.state)
(15, 1)
>>>
```

De la même manière, dans un jeu `undo()` ne renvoie rien, mais modifie si c'est possible, l'attribut `state` du jeu

```
>>> type(jeu)
<class 'allumettes.Matches'>
>>> jeu.state
(15, 1)
>>> print(jeu.undo())
None
>>> jeu.state
(17, 0)
>>> print(jeu.undo()) # sans effet sur le jeu
None
```

```
>>> jeu.state
(17, 0)
>>>
```

- Les paramètres de `__eval_min` et `__eval_max` puisque ce sont des méthodes de la classe `MinMax` elles ont comme premier paramètre `self` qui va permettre d'accéder à toutes les informations utiles et, en particulier à l'état du jeu qui est dans `self.game.state`. Il est **inutile** de garder le paramètre "s", on a juste besoin du paramètre "pf" puisqu'il est modifié à chaque appel. Par ailleurs, pour les appels à ces méthodes on écrira simplement `self.__eval_min(pf -1)` ou `self.__eval_max(pf -1)`

- Pour pouvoir examiner toutes les actions autorisées pour un certain état du jeu, il suffira de faire une boucle `for`

  ```
  for a in self.game.actions:
      self.game.move(a) # change l'état s -> (s,a)
      v = self.__eval_XXX(pf -1) # on appelle min ou max
      traitement de la valeur v
      self.game.undo() # retour à l'état s
  ```

  Important remarquez bien qu'en début de boucle on utilise `move` et qu'en fin de boucle on utilise `undo` – regardez la fiche de cours sur MinMax

- Quand est-ce que l'on est sur une « feuille » ?
  Soit lorsque la partie est terminée `self.game.over()`, soit lorsque l'on a atteint la profondeur maximale autorisée `pf == 0`. L'instruction est

  ```
  if self.game.over() or pf == 0:
     return self.estimation()
  ```

6.3 Mise en œuvre

On sait qu'on a 3 fonctions dont les en-têtes seront

```
def decision(self, state): # renvoie une action
def __eval_min(self, pf:int) -> int: # renvoie une valeur
def __eval_max(self, pf:int) -> int: # renvoie une valeur
```

On sait que `decision` appelera `__eval_min`, que `__eval_min` appelera `__eval_max` et enfin que `__eval_max` appelera `__eval_min`.

7 AlphaBeta

Cet algorithme cherche à éviter de parcourir les branches d'un arbre si on peut prédire que l'information collectée sera sans intérêt pour le calcul. Il s'agit d'une généralisation de l'algorithme de « Branch and Bound »vu au premier semestre, mais qui utilise 2 informations, l'une va servir à

minimiser les explorations `MIN`, l'autre à minimiser les explorations `MAX` La borne `alpha` est une borne inf, c'est-à-dire un majorant des minorants, l'autre est une borne sup, c'est-à-dire un minorant des majorants.
À l'initalisation, il faut que les bornes aient des valeurs à l'extérieur du domaine des évaluations possibles. Comme on sait que l'on a affaire à un jeu à 2 joueurs à « somme nulle », cela signifie que ce que gagne un joueur, l'autre le perd.
Un joueur a un gain maximum défini par `self.WIN`, on a la propriété

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

On peut, dans `decision` faire les bonnes initialisations

```
pf = self.get_value('pf') # on a la profondeur
alpha = - self.WIN - 1
beta = self.WIN + 1
```

Il est **impératif** pour le bon fonctionnement de l'algorithme que pour chaque `move` on fasse `undo` Cela impose que le fonctionnement de la boucle qui parcourt les actions (quelque soit la méthode considérée decision, coupe_alpha, coupe_beta) on ait l'approche suivante

```
for a in self.game.actions:
    self.game.move(a) # changement d'etat
    collect information # récupération d'une valeur
    self.game.undo() # retour à l'état précédent
    traitement information # éventuelle sortie
```

7.1 Exercice de résolution

Cet exercice vise à vérifier que vous avez bien compris le fonctionnement de l'algortihme alpha-beta pour lequel on vous donne l'arbre complet. La fonction d'évaluation autorise des scores compris entre -20 et +20, les scores correspondent au gain (perte) du joueur racine.
Les données sont les mêmes que pour l'exercice de la section 6.1. Les valeurs initiale de $\alpha = -\infty$ et $\beta = +\infty$ lorsqu'on visite pour la première fois le sommet racine.

Exercice 2

1. Quelle est la décision à la racine ?
2. Donnez l'ordre de visite des sommets de l'arbre, l'ordre est obtenu par le premier accès au cours du parcours en profondeur d'abord.
3. Quels sont les sommets qui peuvent modifier la valeur de α ?
4. Quels sont les sommets qui peuvent modifier la valeur de β ?
5. Y-a-t-il des sommets non visités par l'algorithme de l'alpha-beta ?
6. Donnez la valeur finale renvoyée par les sommets visités de l'arbre sauf celles des feuilles, ainsi que les valeurs de α et β lorsqu'on accède pour la première fois au sommet et lorsqu'on quitte le sommet.

7.2 Mise en œuvre

On sait qu'on a 3 fonctions dont les en-têtes seront

```
def decision(self, state): # renvoie une action
def __coupe_alpha(self, pf:int, alpha:int, beta:int) -> int: # renvoie une valeur
def __coupe_beta(self, pf:int, alpha:int, beta:int) -> int: # renvoie une valeur
```

On sait que `decision` appelera `__coupe_alpha`, que `__coupe_alpha` appelera `__coupe_beta` et enfin que `__coupe_beta` appelera `__coupe_beta`.

On peut, pour les boucles sur les actions, soit opter pour une écriture utilisant un `while` comme les algorithmes, soit choisir d'utiliser un `for`. Dans tous les cas, ce qui importe c'est que l'on soit certain que les deux commandes `move(a)` et `undo()` soient effectuées **avant** un éventuel `return` Voici les deux écritures possibles pour la boucle au sein de `__coupe_alpha`

```
...
i = 0
while i < len(self.game.actions) and alpha < beta:
    self.game.move(self.game.actions[i])
    v = self.__coupe_beta(pf-1, alpha, beta)
    self.game.undo()
    if v <= alpha: return alpha
    beta = min(beta, v)
    i = i+1

for a in self.game.actions:
    if alpha >= beta: return beta
    self.game.move(a)
    v = self.__coupe_beta(pf-1, alpha, beta)
    self.game.undo()
    if v <= alpha: return alpha
    beta = min(beta, v)
```