Aide jalon 03

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

Rev. 4 : 13 Avril 2022

1 Liste et Dictionnaire, vademecum

En Python, listes (list) et dictionnaires (dict) sont des containers, c'est-à-dire des structures de stockage, on peut donc les créer, ajouter un élément, supprimer un élément. De plus ces structures sont « itérables », c'est-à-dire que l'on peut les parcourir. Les listes peuvent être vues comme des structures contigües et indexées par des entiers (à partir de 0), contrairement aux dictionnaires.

1.1 Création

```
L = [] # creation d'une liste vide
D = {} # creation d'un dictionnaire vide
```

1.2 Ajout

Deux méthodes d'ajout dans les listes

```
L.append(v) # ajout en fin
L.insert(p, v) # ajout a la position p
```

Pour ajouter dans un dictionnaire

```
D[clef] = v
```

Une clef est une entité qui peut être transformée par une fonction de « hashage » en un nombre. Les clefs qui se prêtent facilement à cette transformation sont les nombres et les chaînes de caractères

```
>>> D['toto'] = 42
>>> D[-2.5] = 'coucou'
```

Une liste, ou un dictionnaire peuvent servir à stocker n'importe quelle information, n'importe quelle structure de données

1.3 Suppression

Pour supprimer une valeur dans une liste on utilise la commande remove, pour supprimer une valeur dans un dictionnaire on utilise la commande del

```
L.remove(v)
del D[clef]
```

1.4 Récupération

Pour récupérer une valeur dans une liste, on utilise simplement la notation `[]`, pour un dictionnaire on dispose de deux méthodes l'une est la notation `[]`, l'autre est la commande `get` qui offre l'avantage lorsque la clef n'existe pas de récupérer une valeur dite valeur par défaut.

```
>>> L = []
>>> for i in range(5, 12): L.append(i)
...
>>> L
[5, 6, 7, 8, 9, 10, 11]
>>> L[2]
7
>>> L[11]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> D = {}
>>> D['toto'] = 42
>>> D[-3] = 'oops'
>>> D[(4,5)] = [1,2,3]
>>> D
{'toto': 42, -3: 'oops', (4, 5): [1, 2, 3]}
>>> D[-3]
'oops'
>>> D.get('un exemple', [8, 9, 10])
[8, 9, 10]
>>> D['un exemple']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'un exemple'
>>> D.get(5, None)
>>> x = D.get(5, None)
>>> type(x)
<class 'NoneType'>
>>> D.get(-3, 7)
'oops'
>>> z = D.get(-3, "je suis -3")
>>> type(z)
<class 'str'>
>>> z
'oops'
```

1.5 Parcours

Dans ce qui suit, `gestion_clef` et `gestion_valeur` désignent n'importe quel code travaillant sur l'information passée en paramètre. On dispose de 3 manières pour parcourir aisément une liste

```
for i in range(len(L)):
    gestion_clef(i)
    gestion_valeur(L[i])

for e in L:
    gestion_valeur(e)

for i,e in enumerate(L):
    gestion_clef(i)
    gestion_valeur(e)
```

Pour les dictionnaires, il y a un parcours standard

```
for k in D:
    gestion_clef(k)
    gestion_valeur(D[k])
```

et on dispose de 3 méthodes particulières

```
D.keys() # renvoie un iterable sur les clefs de D
D.values() # renvoie un iterable sur les valeurs de D
D.items() # renvoie un iterable sur les paires (clef,valeur)
```

Ce qui offre trois types de parcours supplémentaires pour les dictionnaires

```
for k in D.keys():
    gestion_clef(k)
    gestion_valeur(D[k])

for v in D.values():
    gestion_valeur(v)

for k,v in D.items():
    gestion_clef(k)
    gestion_valeur(v)
```

Le premier des 3 parcours, utilisant `D.keys()`, est en fait le parcours standard. L'ordre des 3 parcours utilisant les méthodes des dictionnaires et le même que celui des parcours de listes – si vous avez pris des notes en TD vous retrouverez les similarités que j'avais exposé à l'oral.

Voici un petit exemple dans lequel on souhaite stocker dans un dictionnaire, pour chaque nombre, la valeur de son carré, de son cube et de son inverse :

```
>>> D = {}
>>> for i in range(1, 6):
...     D[i] = {'carre': i*i, 'cube': i**3, 'inverse': round(1/i, 3)}
...
>>> for k in D:
...     print("clef", k, "valeur", D[k])
```

```
...
clef 1 valeur {'carre': 1, 'cube': 1, 'inverse': 1.0}
clef 2 valeur {'carre': 4, 'cube': 8, 'inverse': 0.5}
clef 3 valeur {'carre': 9, 'cube': 27, 'inverse': 0.333}
clef 4 valeur {'carre': 16, 'cube': 64, 'inverse': 0.25}
clef 5 valeur {'carre': 25, 'cube': 125, 'inverse': 0.2}
>>>
```

1.6 Savoir si ...

Pour savoir si un élément ou une clef est dans une liste :

```
if e in L:
    print('yes')
else:
    print('no')
```

```
if i in range(len(L)):
    print('yes')
else:
    print('no')
```

Pour savoir si un élément ou une clef est dans un dictionnaire :

```
if e in D.values():
    print('yes')
else:
    print('no')
```

```
if k in D: # ou if k in D.keys():
    print('yes')
else:
    print('no')
```

Comme on le voit, l'appartenance « naturel » pour une liste est un test sur les valeurs, pour un dictionnaire c'est sur les clefs.

2 NegAlphaBeta_memory

Le but de la mémoire est d'éviter, en premier lieu les recalculs inutiles. Un calcul sera considéré comme inutile si l'information stockée a été obtenu à un profondeur plus grande ou égale, ou bien si le calcul est considéré comme exact.
La première chose que l'on fait quand on « rentre » dans une méthode est de récupérer la clef, et de consulter le contenu de la mémoire après les éventuelles obligations

```
def methode(self, ...):
    # truc obligatoire si methode est decision

    clef = self.game.hash_code
    if clef in self.decision.memory:
        # clef en mémoire, récupération
        memoire = self.decision.memory[clef]
        # traitement avec éventuel arrêt
        if memoire['pf'] >= pf or memoire['exact'] == True:
            # arret
            return memoire['score'] # memoire['best_action']
        #si on est ici, c'est qu'il y a un truc en memoire MAIS
        #pas suffisant pour s'arreter, on peut utiliser le score
        #on peut utiliser best_action (heuristique)
```

```
    # si on est ici c'est qu'on va poursuivre la méthode normale
```

2.1 Comment récupérer la clef du fils ?

Les deux méthodes `decision` et `__cut` effectue une boucle sur les actions

```
for a in self.game.actions:
    self.game.move(a)
    v = - self.__cut(pf-1, ...)
    self.game.undo()
    # exploitation de v
```

Comme la commande `move` modifie l'attribut `state` on va pouvoir obtenir la clef du fils en ajoutant une ligne de code

```
for a in self.game.actions:
    self.game.move(a)
    clef_fils = self.game.hash_code
    v = - self.__cut(pf-1, ...)
    self.game.undo()
    # exploitation de v et de clef_fils
```

2.2 Quelles sont les zones impactées dans le code ?

Le marqueur `# ICI` indique les zones impactées par la gestion de la mémoire dans la version « classique ».

```
def decision(self, state):
    """ the main method """
    self.game.state = state
    if self.game.turn != self.who_am_i:
        print("not my turn to play")
        return None
    beta = self.WIN+1
    alpha = -beta
    score, b_a = alpha, None
    pf = self.get_value('pf')

    # ICI ::DONE::
    parent = self.game.hash_code
    if parent in self.decision.memory:
       _data = self.decision.memory[parent]
       if _data['exact']: return _data['best_action']
       if _data['pf'] >= pf: return _data['best_action']
       alpha = max(alpha, _data['score'])
       if alpha >= beta:
           self.decision.memory[parent]['pf'] = pf
           return _data['best_action']

    for a in self.game.actions:
        self.game.move(a)
        # ICI ::DONE::
        fils = self.game.hash_code
        _ = - self.__cut(pf-1, alpha, beta)
        self.game.undo()
        if _ > score:
            # ICI ::DONE::
            score = _
            b_a = a
            exact = self.decision.memory[fils]['exact']

    # ICI ::DONE::
    self.decision.memory[parent] = { 'pf': pf,
                                     'score': score,
                                     'exact': exact,
                                     'best_action': b_a }
    return b_a

decision.memory = {}

def __cut(self, pf:int, alpha:float, beta:float) -> float:
    """ we use, max thus cut_beta """
    # ICI ::PARTIALLY DONE::
```

```
parent = self.game.hash_code
if parent in self.decision.memory:
   # truc à faire

if pf == 0 or self.game.over():
    _c = 1 if self.who_am_i == self.game.turn else -1
    # ICI ::DONE::
    self.decision.memory[parent] = { 'pf': pf,
                                     'score': _c * self.estimation(),
                                     'exact': self.game.over(),
                                     'best_action': None }
    return _c * self.estimation()

_score = - (self.WIN+1) # - infini
for a in self.game.actions:
    if alpha >= beta:
        # ICI ::TODO::
        return beta
    self.game.move(a)
    # ICI ::DONE::
    fils = self.game.hash_code
    _M = - self.__cut(pf-1, -beta, -alpha)
    self.game.undo()
    # ICI ::DONE::
    if _M > _score
        _score = _M
        b_a = a
        exact = self.decision.memory[fils]['exact']
    alpha = max(_M, score)

# ICI ::DONE::
self.decision.memory[parent] = { 'pf': pf,
                                 'score': _score,
                                 'exact': exact,
                                 'best_action': b_a }
return _score
```

3 Exemples Allumettes

On va suivre, le fonctionnement de l'algorithme sur un cas particulier. On va jouer aux jeux des allumettes, et le but du jeu est de ne **pas** prendre la dernière allumettes. On notera « Xall » la clef `self.game.hash_code` correspondant à la situation où l'on dispose de `X` allumettes. Le joueur `NegAlphaBeta_Memory` doit donner sa décision (combien il doit prendre d'allumettes)

```
>>> from allumettes import Matches
>>> jeu = Matches(13, False)  # ne pas prendre la dernière
```

```
>>> moi = NegAlphaBeta_Memory('moi', jeu, pf=2) # regarde 2 coups en avant
>>> moi.who_am_i = jeu.turn # joue en premier
>>> moi.decision( (4, 4) ) # c'est le 4ème tour de jeu
I am 4All at level 2
I know ...
_______
Insufficient knowledge, work as usual
I am 3All at level 1
I know ...
_______
>> Insufficient knowledge, work as usual with alpha -101, beta 101
I am 2All at level 0
I know ...
_______
>> Insufficient knowledge, work as usual with alpha -101, beta 101
update leaf 2All
I am 1All at level 0
I know ...
2All {'pf': 0, 'score': 0, 'exact': False, 'best_action': None}
_______
>> Insufficient knowledge, work as usual with alpha -101, beta 0
update leaf 1All
I am 0All at level 0
I know ...
2All {'pf': 0, 'score': 0, 'exact': False, 'best_action': None}
1All {'pf': 0, 'score': 0, 'exact': False, 'best_action': None}
_______
>> Insufficient knowledge, work as usual with alpha -101, beta 0
update leaf 0All
update my knowledge 3All
I know ...
2All {'pf': 0, 'score': 0, 'exact': False, 'best_action': None}
1All {'pf': 0, 'score': 0, 'exact': False, 'best_action': None}
0All {'pf': 0, 'score': 100, 'exact': True, 'best_action': None}
3All {'pf': 1, 'score': 0, 'exact': False, 'best_action': 1}
_______
I am 2All at level 1
I know ...
2All {'pf': 0, 'score': 0, 'exact': False, 'best_action': None}
1All {'pf': 0, 'score': 0, 'exact': False, 'best_action': None}
0All {'pf': 0, 'score': 100, 'exact': True, 'best_action': None}
3All {'pf': 1, 'score': 0, 'exact': False, 'best_action': 1}
_______
>> Insufficient knowledge, work as usual with alpha 0, beta 101
I am 1All at level 0
I know ...
2All {'pf': 0, 'score': 0, 'exact': False, 'best_action': None}
1All {'pf': 0, 'score': 0, 'exact': False, 'best_action': None}
0All {'pf': 0, 'score': 100, 'exact': True, 'best_action': None}
```

```
3All {'pf': 1, 'score': 0, 'exact': False, 'best_action': 1}
_______
pf mémoire >= pf actuelle, fin
I am 0All at level 0
I know ...
2All {'pf': 0, 'score': 0, 'exact': False, 'best_action': None}
1All {'pf': 0, 'score': 0, 'exact': False, 'best_action': None}
0All {'pf': 0, 'score': 100, 'exact': True, 'best_action': None}
3All {'pf': 1, 'score': 0, 'exact': False, 'best_action': 1}
_______
pf mémoire >= pf actuelle, fin
update my knowledge 2All
I know ...
2All {'pf': 1, 'score': 0, 'exact': False, 'best_action': 1}
1All {'pf': 0, 'score': 0, 'exact': False, 'best_action': None}
0All {'pf': 0, 'score': 100, 'exact': True, 'best_action': None}
3All {'pf': 1, 'score': 0, 'exact': False, 'best_action': 1}
_______
I am 1All at level 1
I know ...
2All {'pf': 1, 'score': 0, 'exact': False, 'best_action': 1}
1All {'pf': 0, 'score': 0, 'exact': False, 'best_action': None}
0All {'pf': 0, 'score': 100, 'exact': True, 'best_action': None}
3All {'pf': 1, 'score': 0, 'exact': False, 'best_action': 1}
_______
>> Insufficient knowledge, work as usual with alpha 0, beta 101
I am 0All at level 0
I know ...
2All {'pf': 1, 'score': 0, 'exact': False, 'best_action': 1}
1All {'pf': 0, 'score': 0, 'exact': False, 'best_action': None}
0All {'pf': 0, 'score': 100, 'exact': True, 'best_action': None}
3All {'pf': 1, 'score': 0, 'exact': False, 'best_action': 1}
_______
pf mémoire >= pf actuelle, fin
update my knowledge 1All
I know ...
2All {'pf': 1, 'score': 0, 'exact': False, 'best_action': 1}
1All {'pf': 1, 'score': -100, 'exact': True, 'best_action': 1}
0All {'pf': 0, 'score': 100, 'exact': True, 'best_action': None}
3All {'pf': 1, 'score': 0, 'exact': False, 'best_action': 1}
_______
update my knowledge 4All
I know ...
2All {'pf': 1, 'score': 0, 'exact': False, 'best_action': 1}
1All {'pf': 1, 'score': -100, 'exact': True, 'best_action': 1}
0All {'pf': 0, 'score': 100, 'exact': True, 'best_action': None}
3All {'pf': 1, 'score': 0, 'exact': False, 'best_action': 1}
4All {'pf': 2, 'score': 100, 'exact': True, 'best_action': 3}
_______
```

```
3 # ceci est la décision finale
>>>
```

```
>>> from allumettes import Matches
>>> jeu = Matches(13, True)  # prendre la dernière
>>> NegAlphaBeta_Memory.decision.memory = {} # flush memory
>>> moi = NegAlphaBeta_Memory('moi', jeu, pf=3) # regarde 3 coups en avant
>>> moi.who_am_i = jeu.turn # joue en premier
>>> moi.decision( (4, 4) ) # c'est le 4ème tour de jeu
I am 4All at level 3
I know ...
_______
Insufficient knowledge, work as usual
I am 3All at level 2
I know ...
_______
>> Insufficient knowledge, work as usual with alpha -101, beta 101
I am 2All at level 1
I know ...
_______
>> Insufficient knowledge, work as usual with alpha -101, beta 101
I am 1All at level 0
I know ...
_______
>> Insufficient knowledge, work as usual with alpha -101, beta 101
update leaf 1All
I am 0All at level 0
I know ...
1All {'pf': 0, 'score': 0, 'exact': False, 'best_action': None}
_______
>> Insufficient knowledge, work as usual with alpha -101, beta 0
update leaf 0All
update my knowledge 2All
I know ...
1All {'pf': 0, 'score': 0, 'exact': False, 'best_action': None}
0All {'pf': 0, 'score': -100, 'exact': True, 'best_action': None}
2All {'pf': 1, 'score': 100, 'exact': True, 'best_action': 2}
_______
I am 1All at level 1
I know ...
1All {'pf': 0, 'score': 0, 'exact': False, 'best_action': None}
0All {'pf': 0, 'score': -100, 'exact': True, 'best_action': None}
2All {'pf': 1, 'score': 100, 'exact': True, 'best_action': 2}
_______
>> Insufficient knowledge, work as usual with alpha 0, beta 100
I am 0All at level 0
I know ...
1All {'pf': 0, 'score': 0, 'exact': False, 'best_action': None}
0All {'pf': 0, 'score': -100, 'exact': True, 'best_action': None}
```

```
2All {'pf': 1, 'score': 100, 'exact': True, 'best_action': 2}
_______
pf mémoire >= pf actuelle, fin
update my knowledge 1All
I know ...
1All {'pf': 1, 'score': 100, 'exact': True, 'best_action': 1}
0All {'pf': 0, 'score': -100, 'exact': True, 'best_action': None}
2All {'pf': 1, 'score': 100, 'exact': True, 'best_action': 2}
_______
I am 0All at level 1
I know ...
1All {'pf': 1, 'score': 100, 'exact': True, 'best_action': 1}
0All {'pf': 0, 'score': -100, 'exact': True, 'best_action': None}
2All {'pf': 1, 'score': 100, 'exact': True, 'best_action': 2}
_______
score exact, fin
update my knowledge 3All
I know ...
1All {'pf': 1, 'score': 100, 'exact': True, 'best_action': 1}
0All {'pf': 0, 'score': -100, 'exact': True, 'best_action': None}
2All {'pf': 1, 'score': 100, 'exact': True, 'best_action': 2}
3All {'pf': 2, 'score': 100, 'exact': True, 'best_action': 3}
_______
I am 2All at level 2
I know ...
1All {'pf': 1, 'score': 100, 'exact': True, 'best_action': 1}
0All {'pf': 0, 'score': -100, 'exact': True, 'best_action': None}
2All {'pf': 1, 'score': 100, 'exact': True, 'best_action': 2}
3All {'pf': 2, 'score': 100, 'exact': True, 'best_action': 3}
_______
score exact, fin
I am 1All at level 2
I know ...
1All {'pf': 1, 'score': 100, 'exact': True, 'best_action': 1}
0All {'pf': 0, 'score': -100, 'exact': True, 'best_action': None}
2All {'pf': 1, 'score': 100, 'exact': True, 'best_action': 2}
3All {'pf': 2, 'score': 100, 'exact': True, 'best_action': 3}
_______
score exact, fin
update my knowledge 4All
I know ...
1All {'pf': 1, 'score': 100, 'exact': True, 'best_action': 1}
0All {'pf': 0, 'score': -100, 'exact': True, 'best_action': None}
2All {'pf': 1, 'score': 100, 'exact': True, 'best_action': 2}
3All {'pf': 2, 'score': 100, 'exact': True, 'best_action': 3}
4All {'pf': 3, 'score': -100, 'exact': True, 'best_action': 1}
_______
1 # ceci est la décision finale
```

3.1 Je ne comprends pas ...

Pour savoir si votre code est opérationnel, il y a d'une part les testcodes fournis (`test_clever`) mais aussi `main_tests`. L'inconvénient c'est que « vous ne comprenez pas » disons plutôt que vous ne savez pas quoi faire pour y remédier.

- Le problème des `testcode` c'est qu'ils inhibent les messages d'erreurs. Pour les obtenir, il faut que vous recopiez les codes dans le `shell`. Les étapes sont toujours les mêmes, importer le jeu, créer le jeu, vider la mémoire, créer le joueur, dire que c'est le premier à jouer, lui demander quelle est sa décision pour un cas particulier. Par exemple je veux tester mon joueur sur le jeu des allumettes, dans une partie où il ne doit pas prendre la dernière et lorsqu'il doit choisir avec devant lui 4 allumettes

```
from allumettes import Matches
jeu = Matches(13, False) # ne pas prendre la dernière
NegAlphaBeta_Memory.decision.memory = {}
moi = NegAlphaBeta_Memory('tyty', jeu, pf=3) # profondeur max 3
moi.who_am_i = jeu.turn
moi.decision( (4, 4) ) # il y a 4 allumettes et c'est le tour 4
```

  On **sait** que le meilleur coup est de prendre 3 allumettes ; par contre si le jeu est de prendre la dernière allumette on fera

```
from allumettes import Matches
jeu = Matches(13, True) # prendre la dernière
NegAlphaBeta_Memory.decision.memory = {}
moi = NegAlphaBeta_Memory('tyty', jeu, pf=3) # profondeur max 3
moi.who_am_i = jeu.turn
moi.decision( (4, 4) ) # il y a 4 allumettes et c'est le tour 4
```

  Dans ce cas, on **sait** qu'il n'y a pas d'action gagnante et que le joueur prendra 1 allumette

- Le problème des tests complexes (`main_tests`), c'est qu'il y a plusieurs tests et qu'il faut **lire** ce qui est écrit. Quand il y a un échec vous avez un message tel que celui-ci :

```
FAIL: test_subkeys_values (tests.test_negalpha_mem.TestMemory)
check data values for DO NOT Take the last
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users[redacted]\AppData\Local\Programs\Python\Python39\lib\unittest\mock.
py", line 1337, in patched
    return func(*newargs, **newkeywargs)
  File "C:\Users[redacted]Bureau\MIASHS\L3 MIASHS\SEMESTRE 6\Intelligence
artificielle\Projet_IA\Code\tests\test_negalpha_mem.py", line 327, in test_subke
ys_values
    self.assertEqual(_1, 3, "decision should return 3, found {}".format(_1))
AssertionError: 1 != 3 : decision should return 3, found 1
```

  Ce message indique que l'erreur est due à un test dans le fichier `test_negalpha_mem` dans la classe `TestMemory` Que le but est de **ne pas prendre la dernière** et que la décision fournie a été « 1 » tandis que la réponse attendue était « 3 ».
  90% des erreurs de la classe `NegAlphaBeta_Memory` sont des erreurs de contenus de la mémoire. Votre seul recours est donc de tester ce contenu, en vérifiant que ce qui est produit par votre code correspond à ce qui est attendu. Pour cela, dans le shell, après les lignes de code sur la prise de décision, il faut que vous oscultiez le contenu de la mémoire en utilisant le code suivant :

```
for k in moi.decision.memory:
  print(k, moi.decision.memory[k])
```

Normalement, si vous avez recopié la classe `NegAlphabeta` les **seules** erreurs ne devraient être que le contenu de la mémoire. Le test mis en œuvre dans la classe `TestMemory` correspondent à la situation suivante :

```
>>> from allumettes import Matches
>>> jeu = Matches(13, False) # ne pas prendre la derniere
>>> NegAlphaBeta_Memory.decision.memory = {} # RAZ memoire
>>> moi = NegAlphaBeta_Memory('x', jeu, pf=3)
>>> moi.who_am_i = jeu.turn
>>> moi.decision((4,4)) # 4 allumettes
3
>>> for k in moi.decision.memory:
...     print(k, moi.decision.memory[k])
...
>>>
0 {'pf': 0, 'score': 100, 'exact': True, 'best_action': None}
1 {'pf': 2, 'score': -100, 'exact': True, 'best_action': 1}
2 {'pf': 2, 'score': 0, 'exact': False, 'best_action': 1}
3 {'pf': 2, 'score': 0, 'exact': False, 'best_action': 1}
4 {'pf': 3, 'score': 100, 'exact': True, 'best_action': 3}
```

4 IterativeDeepening

Pour cette classe nous allons commencer par dupliquer la classe `NegAlphaBeta_Memory` et changer le nom de la classe. Puis nous allons renommer la méthode `decision` par exemple `__old_decision` Ce faisant nous avons maintenant 3 méthodes dans la classe `IterativeDeepening`

1. `decision` C'est la méthode principale de la classe, son rôle est de collecter les paramètres externes, mettre à jour la variable `self.game.state` puis de faire une boucle qui appelle à chaque itération `__old_decision` avec les bons paramètres

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

        # the parameters to retrieve
        pf = self.get_value('pf')
        second = self.get_value('secondes')
        depth = ...
        # alpha, beta
        _bound = self.WIN +1
        _start = time.time()
```

```
        while depth <= pf and time.time() - _start < second:
            _a = self.__old_decision(depth, -_bound, +_bound)
            # print("elapsed {:.2f} depth = {}"
            #       "".format(time.time() - _start, depth))
            depth += 1
        return _a
```

2. `__old_decision` On va devoir changer sa signature, son rôle est simplement de renvoyer la meilleure action pour un calcul à une profondeur fixée

   ```
   def __old_decision(self, pf:int, alpha:float, beta:float):
   ```

   Bien entendu, on va évacuer du code de `__old_decision` tout ce qui ne sert pas comme : vérifier le tour de jeu, demander quelle est la profondeur, initialiser alpha et beta, ou encore regarder si l'état est en mémoire. Dit autrement on ne garde que la boucle sur les actions et la sauvegarde en mémoire avant de retourner l'action choisie.

3. `__cut` Aucun changement par rapport à la méthode de la classe `NegAlphaBeta_Memory`

Attention La détection des feuilles de l'arbre de jeu se fait lorsque `pf == 0`, si vous commencez l'iterative deepening avec la variable `depth` à **0**, vous allez avoir des profondeurs négatives et donc un risque de boucle infinie.

5 Méthodes de Monte-Carlo : la méthode `simulation`

La classe `Player` offre une méthode `simulation` qui dispose de deux paramètres un entier qui est le nombre de simulations à effectuer, et un booléen qui spécifie si les calculs sont fait du point de vue du joueur racine, ou du point de vue du joueur local. Par défaut les calculs sont fait du point de vue du joueur racine.

La méthode a la signature suivante :

```
def simulation(self, n:int=10, rootPlayer:bool=True) -> list:
    """ require n > 0, provides [victory, defeat, draw] """
```

Toutes les classes qui héritent de `Player` dispose de cette méthode, ce qui va nous permettre de tester facilement les effets du booléen.

Attention le nombre de simulations attendu est un entier strictement positif.

```
>>> from allumettes import Matches
>>> jeu = Matches(13, False)
>>> from sol_j02 import Randy
>>> moi = Randy('mmc', jeu)
>>> moi.who_am_i = jeu.turn
>>> moi.game.state = (4,4) # 4 allumettes au tour 4
>>> for a in moi.game.actions:
...     moi.game.move(a)
...     print("point de vue racine {} apres action {}"
              "".format(moi.who_am_i, a),
```

```
              moi.simulation(1000))
...     print("point de vue joueur local {} apres action {}"
              "".format(moi.game.turn, a),
              moi.simulation(1000, False))
...     moi.game.undo()
...
point de vue racine 0 apres action 1 [516, 484, 0]
point de vue joueur local 1 apres action 1 [492, 508, 0]
point de vue racine 0 apres action 2 [528, 472, 0]
point de vue joueur local 1 apres action 2 [483, 517, 0]
point de vue racine 0 apres action 3 [1000, 0, 0]
point de vue joueur local 1 apres action 3 [0, 1000, 0]
>>> moi.simulation(3000)
[1974, 1026, 0]
>>> moi.simulation(3000, False)
[1971, 1029, 0]
```

Les premières lignes vous sont désormais familières, on crée un jeu d'allumettes, avec au départ 13 allumettes et l'objectif et de ne pas prendre la dernière allumette. On crée ensuite un joueur et il est celui qui joue les coups pairs. On place le joueur dans la situation où il reste 4 allumettes et qu'on est au tour 4. Il est donc à la fois le joueur racine et le joueur qui a « localement » le trait. On fait ensuite une boucle sur les actions, on joue l'action on fait deux calculs l'un avec le point de vue du joueur racine, l'autre avec le point de vue du joueur qui a localement le trait – comme on a joué un coup le joueur qui a le trait après le coup, est l'adversaire du joueur racine, il a donc un point de vue opposé.
Comme les résultats sont les issues de parties aléatoires, après avoir pris 3 allumettes, le score est sans appel ; du point de vue du joueur racine il est sûr de gagner, du point de vue du joueur confronté à la situation (4 - 3) allumettes il est certain de perdre.
Pour les deux autres alternatives de jeu, les tirages aléatoires montrent une situation légèrement favorable au joueur racine ; 516 contre 484 et 528 contre 472.
En cumulant tous les compteurs on peut estimer si le joueur ayant 4 allumettes est dans une situation favorable ou défavorable. L'évaluation donne [516+528+1000, 484+472+0, 0], soit [2044, 956, 0] ce qui donne environ $\langle \frac{2}{3}, \frac{1}{3} \rangle$.
C'est ce qui ressort des deux dernières lignes de code, vous noterez que demander une estimation du point de vue du joueur racine ou du point de vue du joueur qui a localement le trait donne le même ordre de grandeur, puisqu'il s'agit du même joueur. Les disparités constatées ne sont dûes qu'au caractère aléatoire des simulations.

Le script présenté est très proche du travail qui est demandé pour la classe `Randy_MC`

5.1 Randy_MC

Cette classe sans mémoire, utilise le paramètre `nbSim`,

```
class Randy_MC(Player):
    def decision(self, state):
        self.game.state = state
        if self.game.turn != self.who_am_i:
```

```
            print("not my turn to play")
            return None

        _nSims = self.get_value('nbSim')
        if _nbSims < 1: return None

        _best_score = -1.5 # scoring renvoie un nombre entre -1 et +1
        _best_action = None
        for a in self.game.actions:
            self.game.move(a)
            _scores = self.simulation(_nSims)
            self.game.undo()
            _e = # calcul du scoring
            if _best_action is None or _e > _best_score:
               # update

        return _best_action
```

5.2 UCB

Cette classe, sans mémoire, utilise le paramètre `nbSim`. La différence avec la classe `Randy_MC` est de répartir différemment les simulations. Pour `Randy_MC` on effectue `nbSim` simulations pour chaque action, soit un total de nbSim × self.game.actions.
Ici il faudra travailler en 2 temps, premier temps, pour chaque action on fait **une** simulation. Dans un second temps on effectue une boucle de longueur (nbSim − 1) × self.game.actions dans laquelle on choisit l'action ayant la meilleure « utilité UCB », calculée à l'aide de la formule

$$\text{utilite}(a) + 0.3 \times \sqrt{\frac{\log(n)}{n_a}}$$

Où utilite(a) est obtenue grâce à la fonction `scoring`

```
class UCB(Player):
    def decision(self, state):
        self.game.state = state
        if self.game.turn != self.who_am_i:
            print("not my turn to play")
            return None

        _scores = {}
        _nSims = self.get_value('nbSim')
        if _nbSims < 1: return None

        for a in self.game.actions:
            self.game.move(a)
            _scores[a] = self.simulation(1) # stockage
            self.game.undo()
```

```
        for i in range((_nbSims -1)*len(self.game.actions)):
            # déterminer b qui maximise formule UCB
            self.game.move(b)
            _resultat = self.simulation(1)
            self.game.undo()
            # mettre à jour _scores[b] avec _resultat

        # déterminer a qui maximise formule UCB
        return a
```

Il n'y a aucune obligation à stocker les informations dans un dictionnaire, on peut tout à fait stocker les compteurs dans une liste.

```
class UCB(Player):
    def decision(self, state):
        self.game.state = state
        if self.game.turn != self.who_am_i:
            print("not my turn to play")
            return None

        _scores = []
        _nSims = self.get_value('nbSim')
        if _nbSims < 1: return None

        for a in self.game.actions:
            self.game.move(a)
            _scores.append(self.simulation(1)) # stockage
            self.game.undo()

        for i in range((_nbSims -1)*len(self.game.actions)):
            # déterminer idx qui maximise formule UCB
            b = self.game.actions[idx]
            self.game.move(b)
            _resultat = self.simulation(1)
            self.game.undo()
            # mettre à jour _scores[idx] avec _resultat

        # déterminer a qui maximise formule UCB
        return a
```

Je vous encourage à écrire une méthode qui reçoit en entrée la structure de stockage (liste ou dictionnaire) contenant les compteurs et qui renvoie l'action (dictionnaire) ou l'index (liste) correspondant à la plus grande valeur pour l'« utilité UCB ».

5.3 NegAlphaBeta_MC

La seule précaution à prendre pour le calcul au niveau des feuilles, est de choisir entre une simulation du point de vue du joueur racine `self.simulation(nbSim)` ou du joueur qui a le trait `self.simulation(nbSim, False)` avant de renvoyer le score calculé à l'aide de la fonction `scoring(...)`

5.4 UCT

L'idée de l'algorithme `UCT` consiste à construire un arbre des simulations en ne développant que la branche qui semble la plus prometteuse d'après les simulations effectuées. Pour fonctionner cet algorithme a besoin d'une mémoire dont les clefs sont les `hash_code` des stituations de jeu (donc même principe que pour l'algorithme `NegAlphaBeta_Memory`) et les valeurs sont soit

1. Un dictionnaire de compteurs, indexés par l'action
2. Une liste de compteurs dont l'ordre dépend des actions

Comme l'interprétation des compteurs dépend du joueur (voir la section 5), il est important de savoir si on fait les calculs du point de vue du joueur racine, ou du point de vue du joueur qui a le trait.

- Si vous avez choisi de faire en fonction du joueur racine, le gros avantage c'est que comme tout le monde a le même point de vue, les mises à jour de compteurs sont faciles :

  ```
  _cpt # désigne les compteurs pour une action
  _new # désigne les informations à intégrer
  _cpt = [x+y for x,y in zip(_cpt,_new)]
  ```

  Le **gros** désavantage c'est que votre joueur ne sera pas très efficace du fait que dans un arbre de jeu, une même situation peut se rencontrer à différentes profondeurs et donc que les compteurs des simulations entre en conflit. Par exemple j'ai **0** allumette, je suis le premier joueur j'ai gagné, je suis le second joueur j'ai gagné et donc le premier joueur a perdu. (cf fin document, section 5.4.2)

- Si vous avez choisi de faire en fonction du joueur qui a la main, le gros désavantage c'est qu'il faut inverser le rôle des points de victoires et de défaites lors des mises à jour

  ```
  _cpt # désigne les compteurs pour une action
  _new # désigne les informations à intégrer
  _cpt = [_cpt[0]+_new[1], _cpt[1]+_new[0], _cpt[2]+_new[2]]
  ```

  que l'on peut facilement automatiser avec une fonction pour éviter les erreurs

  ```
  def opposite_sum(me:list, you:list) -> list:
      """ what I win is what you loose """
      return [me[0]+you[1], me[1]+you[0], me[2]+you[2]]
  ```

  Le **gros** avantage c'est que l'on dispose maintenant d'un joueur qui peut exploiter les données quelque soit son rôle (premier ou second joueur)

Notre classe va recevoir du constructeur, deux informations, la profondeur d'exploration `pf` et le nombre de simulations à effectuer lorsqu'on rencontre un nouvel état `nbSim` La méthode `decision` est donnée dans la section 5.3.1 page 8 du descriptif du jalon. Vous constatez que cette méthode fait appel à deux méthodes

- `self.__simulations` qui reçoit en paramètre la profondeur et le nombre de simulations. Son objectif est de développé une branche de l'arbre de jeu et de remplir la mémoire avec des compteurs. Son rôle est de garantir que les informations soient remontées (en réalité, on va mettre à jour les informations dans la mémoire) lorsque le calul est terminé.

- `self.__ucb_policy` sans paramètre, son rôle est de faire le calcul de l'utilité « ucb », et de déterminer quelle est l'action ayant le plus fort score.

5.4.1 Pas à pas

Prenons le jeu des allumettes, on démarre avec une mémoire vierge. Le but est d'explorer jusqu'à la profondeur 2, et de faire 10 simulations. On se place du point de vue du joueur **racine**.

Ci-dessous, le code que nous allons exploiter. La ligne 5 c'est la consultation de la mémoire. La ligne 6 c'est la variable qui tient à jour les nouvelles simulations

```
1  def __simulations(pf, nbSim):
2      """ keep local creation, get next level counters
3          invert win/loss when updating counters
4      """
5      get the old scores for each action
6      _sum = [0,0,0]
7      if pf == 0; return _sum
8
9      if self.game.over():
10        get counters for None
11        increase counters for None by nbSim
12        increase _sum by nbSim
13        return _sum
14
15     for a in self.game.actions:
16         if a has counters > 0: continue # no new simulation
17         make nbSim simulations starting by a until game.over()
18
19         increase counters for a
20         increase _sum
21     # all actions have at least one simulation
22     # now performs the best action and get result
23     next_action = self.__ucb_policy()
24     self.game.move(next_action)
25     _result = self.__simulations(pf-1, nbSim)
26     self.game.undo()
27     update counters for next_action with _result
28     update _sum with _result
29
30     return _sum
```

1. On consulte la mémoire, elle est vide, les lignes 7 à 13 sont ignorées. On effectue une boucle (lignes 15 à 20), sur chaque action pour 7 allumettes, à la profondeur 2

```
parent = self.game.hash_code
cpt = {}
for a in self.game.actions:
    self.game.move(a)
    cpt[a] = self.simulation(10)
    self.game.undo()
    for k in range(3): _sum[k] = cpt[a]
```

Supposons que `cpt` contienne

```
{1: [2, 8, 0], 2: [8, 2, 0], 3: [4, 6, 0]}
```

On aura donc dans `_sum` les valeurs `[14, 16, 0]` On fait appel à `ucb_policy` qui après évaluation dit, que l'action à privilégier est **2**. (lignes 23, 24)

2. L'étape suivante va consister à jouer « prendre 2 allumettes » et à appeler `simulations`
   On consulte la mémoire pour 5 allumettes à la profondeur 1. On effectue les lignes 15 à 20 ; supposons qu'à cette étape on trouve

   ```
   {1: [8, 2, 0], 2: [2, 8, 0], 3: [5, 5, 0]}
   ```

   On aura donc dans `_sum` les valeurs `[15, 15, 0]` On fait appel à `ucb_policy` qui après évaluation dit, que l'action à privilégier est **1**. (lignes 23, 24)

3. L'étape suivante va consister à jouer « prendre 1 allumette » et à appeler `simulations`
   On consulte la mémoire pour 4 allumettes à la profondeur 0
   Cette fois-ci on effectue les lignes 5 à 7.
   Comme on est à la profondeur 0, pas de simulation possible, on renvoie l'information `[0,0,0]`

4. On revient donc à 5 allumettes. Qui a reçu le résultat `[0,0,0]` (ligne 25)
   On « déjoue » grâce à `self.game.undo()` (ligne 26)
   On doit faire les lignes 27 et 28. La ligne 27 a pour conséquence le **stockage** dans `self.decision.memory[parent]` des **compteurs** pour chaque action

   ```
   {5: {1: [8, 2, 0], 2: [2, 8, 0], 3: [5, 5, 0]}}
   ```

   Ou sous forme de liste

   ```
   {5: [[8, 2, 0],[2, 8, 0],[5, 5, 0]]}
   ```

   et on renvoie `[15, 15, 0]`, ce qui nous ramène dans l'état 7 allumettes à la ligne 25
   On effectue les lignes 26, 27 et 28, qui provoque le changement d'information en mémoire

   ```
   {5: {1: [8, 2, 0], 2: [2, 8, 0], 3: [5, 5, 0]},
    7: {1: [2, 8, 0], 2: [23, 17, 0], 3: [4, 6, 0]}}
   ```

   Qui à sont tour renvoie l'information `[29, 31, 0]` Cette information n'est pas utilisée dans `decision` ; après avoir appeler `self.__simulations(2, 10)` On fait appel à `self.__ucb_policy` qui va exploiter la mémoire pour indiquer qu'il faut jouer l'action **2**

```
============= ucb_policy =============
1 -0.4250406434769509
2 0.774959356523049
3 -0.025040643476950925

best_action is : 2
_______
```

Attention dans cet exemple, traité à la main, les compteurs sont à comprendre de la manière suivante

- Pour 7 allumettes, le joueur **racine** s'il prend une allumette gagne 2 fois, perd 8 fois ; s'il prend 2 allumettes il gagne 8 fois et perd 2 fois ; s'il prend 3 allumettes il gagne 4 fois et perd 6 fois. Il choisit donc de prendre 2 allumettes
- Pour 5 allumettes, celui qui a le trait (en l'occurence l'adversaire du joueur racine) s'il prend 1 allumette le joueur **racine** gagne 8 fois et perd 2 fois ; s'il prend 2 allumettes le joueur **racine** gagne 2 fois et perd 8 fois ; s'il prend 3 allumettes le joueur **racine** gagne aussi souvent qu'il ne perd. Du point de vue du joueur **racine**, il est souhaitable que son adversaire prenne 1 allumette – je ne suis pas certain que son adversaire est vraiment envie de lui faire plaisir ...

5.4.2 Comparaisons entre UCT_max et UCT_negamax

J'ai implémenté les deux versions, `UCT_max` tient à jour les compteurs du point de vue du joueur **racine** mais tient compte du joueur qui a le trait pour choisir l'action. La version `UCT_negamax` est la version dans laquelle les compteurs sont du point de vue du joueur qui a le trait.

Deux exécutions successives avec comme paramètre une profondeur d'exploration de **5** et un nombre de simulations fixé à **50** contre 6 adversaires. Les joueurs utilisent des techniques de Monte-Carlo avec différents raffinements et en utilisant ou pas une mémoire (`UCB` et `Randy_MC` sont les classes que vous avez à réaliser).

```
UCT_max UCT_negamax (0, 7)
UCT_max UCB (8, 0)
UCT_max Randy (0, 7)
UCT_max Randy_MC (6, 0)
UCT_max Randyy_MC (8, 0)
UCT_max Randyyy_MC (0, 7)
========================
UCT_negamax UCB (8, 0)
UCB UCT_negamax (0, 9)
UCT_negamax Randy (10, 0)
Randy UCT_negamax (0, 7)
UCT_negamax Randy_MC (8, 0)
Randy_MC UCT_negamax (8, 0)
UCT_negamax Randyy_MC (0, 7)
```

```
Randyy_MC UCT_negamax (8, 0)
UCT_negamax Randyyy_MC (0, 9)
Randyyy_MC UCT_negamax (6, 0)
```

Dans la première sortie, UCT_max joue toujours en premier, qu'il a gagné 3 fois et perdu 3 fois – soit la moitié des rencontres gagnées.
Le joueur UCT_negamax a gagné 3 fois en tant que premier joueur et gagné 2 fois en tant que second joueur – soit 5 rencontres sur 12 qui sont remportées.

```
UCT_max UCT_negamax (0, 7)
UCT_max UCB (6, 0)
UCT_max Randy (6, 0)
UCT_max Randy_MC (0, 7)
UCT_max Randyy_MC (0, 7)
UCT_max Randyyy_MC (6, 0)
========================
UCT_negamax UCB (8, 0)
UCB UCT_negamax (6, 0)
UCT_negamax Randy (6, 0)
Randy UCT_negamax (0, 9)
UCT_negamax Randy_MC (0, 7)
Randy_MC UCT_negamax (8, 0)
UCT_negamax Randyy_MC (6, 0)
Randyy_MC UCT_negamax (0, 7)
UCT_negamax Randyyy_MC (6, 0)
Randyyy_MC UCT_negamax (0, 7)
```

Dans la seconde sortie, UCT_max a gagné 3 fois et perdu 3 fois – toujours le même ratio, tandis que UCT_negamax a gagné 4 fois en premier joueur, et 3 fois en second joueur – soit 7 victoires sur 12 rencontres.

La sortie suivante est simplement un affichage du contenu de la mémoire pour les deux versions UCT_max et UCT_negamax

```
============ UCT_max ============
> 0 {None: [550, 100, 0]}
> 1 {1: [450, 100, 0]}
> 2 {1: [350, 0, 0], 2: [50, 50, 0]}
> 3 {1: [75, 25, 0], 2: [200, 0, 0], 3: [100, 50, 0]}
> 4 {1: [124, 26, 0], 2: [22, 28, 0], 3: [0, 100, 0]}
> 5 {1: [81, 19, 0], 2: [128, 22, 0], 3: [277, 73, 0]}
> 6 {1: [121, 29, 0], 2: [76, 174, 0], 3: [88, 112, 0]}
> 7 {1: [25, 25, 0], 2: [71, 29, 0], 3: [29, 21, 0]}
> 8 {1: [24, 26, 0], 2: [22, 28, 0], 3: [313, 137, 0]}
> 9 {1: [20, 30, 0], 2: [152, 98, 0], 3: [170, 180, 0]}
> 10 {1: [28, 22, 0], 2: [21, 29, 0], 3: [22, 28, 0]}
```

```
> 11 {1: [32, 18, 0], 2: [315, 335, 0], 3: [30, 20, 0]}
> 13 {1: [21, 29, 0], 2: [406, 394, 0], 3: [22, 28, 0]}
==================================

============= UCT_negamax =============
> 0 {None: [1050, 0, 0]}
> 1 {1: [0, 1100, 0]}
> 2 {1: [400, 0, 0], 2: [0, 50, 0]}
> 3 {1: [24, 26, 0], 2: [450, 0, 0], 3: [0, 50, 0]}
> 4 {1: [25, 25, 0], 2: [26, 24, 0], 3: [200, 0, 0]}
> 5 {1: [20, 80, 0], 2: [106, 344, 0], 3: [76, 274, 0]}
> 6 {1: [124, 26, 0], 2: [12, 38, 0], 3: [26, 74, 0]}
> 7 {1: [21, 29, 0], 2: [233, 67, 0], 3: [15, 35, 0]}
> 8 {1: [111, 89, 0], 2: [25, 25, 0], 3: [25, 25, 0]}
> 9 {1: [173, 177, 0], 2: [29, 21, 0], 3: [26, 24, 0]}
> 10 {1: [251, 249, 0], 2: [27, 23, 0], 3: [78, 72, 0]}
> 11 {1: [325, 325, 0], 2: [20, 30, 0], 3: [28, 22, 0]}
> 12 {1: [28, 22, 0], 2: [27, 23, 0], 3: [21, 29, 0]}
> 13 {1: [18, 32, 0], 2: [22, 28, 0], 3: [73, 77, 0]}
======================================
```

Le joueur `UCT_max` a fait 11 séries de simulations en situation gagnante (0 allumettes) et 2 séries en situation perdante. Par contre, le joueur `UCT_negamax` qu'il ait joué en premier ou en second a toujours considéré la situation (0 allumettes) comme gagnante.

Si on examine les compteurs du joueur `UCT_negamax` on peut obtenir des informations sur ce que l'on nomme en théorie des jeux une situation gagnante et une situation perdante.
Une situation est dite gagnante, s'il existe un coup qui conduit à une situation perdante pour l'adversaire. Une situation est dite perdante si quelque soit l'action on aboutit à une situation gagnante pour l'adversaire.

1. La situation « **0** allumette » est une situation gagnante
2. La situation « **1** allumette » est une situation perdante
3. Les situations « **2** allumettes », « **3** allumettes » et « **4** allumettes » sont gagnantes
4. La situation « **5** allumettes » est une situation perdante

Jusqu'à **7** allumettes, le nombre de simulations permet de déterminer de façon certaine si une situation est gagnante ou perdante. Cette analyse est ce que l'on appelle une analyse rétrograde, très utile pour pouvoir définir une fonction d'évaluation performante mais cela est une autre histoire ... que peut-être certains d'entre vous aborderont avec la classe `Ultimate`

CECI EST LA DERNIÈRE MISE À JOUR, SAUF DÉTECTION DE TYPOS