Jeu de Nim

December 10, 2024

1 Présentation

On implémente ici les notions d'*attracteur* et de recherche *min-max* adaptés au problème du jeu de Nim (ou jeu des allumenttes).

Dans ce jeu à 2 joueurs, la configuration initiale est un tas d'allumettes. Chaque joueur retire entre 1 et n allumettes du tas. Le gagnant est celui qui retire les deux allumettes.

DERNIERES ALLUMETTES

2 Outils

On pose init=17, le contenu initial du tas

On utilise les énumérations python. On crée d'abord un type `Player` :

[2]:
```
from enum import Enum

class Player(Enum):
    one=1
    two=2

J1=Player.one
J2=Player.two
type (J1), J1.value, J1.name
```

[2]:
```
(<enum 'Player'>, 1, 'one')
```

[3]:
```
J1 is Player.one
```

[3]:
```
True
```

Les états sont des tuples (joueur, nombre d'allumettes).

2.0.1 Q1

Ecrire la fonction `other(p)` qui prend en paramètre un joueur et renvoie l'autre joueur.

[34]:
```
other(Player.one)
```

[34]:
```
<Player.two: 2>
```

2.0.2 Q2

Ecrire la fonction `player(e:(Player,int))->Player` qui renvoie le joueur auquel appartient l'état.

[36]:
```
e = (Player.one,17)
p = player(e)
print("{}".format(p.name))
```

```
one
```

De même, écrire la fonction `quantity(e:(Player,int))->int` qui renvoie le nombre d'allumettes du tas.

[38]:
```
e = (Player.one,17)
nb = quantity(e)
print("{}".format(nb))
```

```
17
```

2.0.3 Q3

Ecrire la fonction `terminal(e:(Player,int))->bool` qui indique si un état est terminal.

[40]:
```
e = (Player.one,17)
terminal(e)
```

[40]:
```
False
```

[41]:
```
e = (Player.two,0)
terminal(e)
```

[41]:
```
True
```

2.0.4 Q4

De même, écrire une fonction `authorized(e:(Player,int))->bool` qui indique si un état est possible (nombre d'allumettes positif par exemple)

[83]:
```
authorized((Player.one,-1)),authorized((Player.two,17)),authorized((Player.
↪one,init))
```

[83]:
```
(False, False, True)
```

auxquelles

On définit une exception qui sera soulevée par des fonctions n'acceptant que des états terminaux et auxquels on communique un état qui ne l'est pas :

[84]:
```
class Not_Terminal(Exception):
    def __str__(self):
        return "Etat non terminal"
```

A METTRE AU DEBUT

On définit maintenant la variable globale `N` : elle indique le nombre maximum d'allumettes qu'on peut retirer du tas:

[85]:
```
N=3
```

Enfin, on définit la varaible globale `init` : elle indique le nombre initial d'allumettes dans le tas.

[86]:
```
init = 17
```

2.0.5 Q5

Ecrire la fonction `outcome(e:(Player,int)->Player` qui soulève une exception `Not_Terminal` si l'état `e` n'est pas terminal. Elle renvoie le joueur gagnant sinon.

Remarque : dans certains jeux, il existe des états de match nul. Dans ce cas, il faudrait adapter la fonction pour qu'elle renvoie `None` en cas de match nul.

[88]:
```
e = (Player.two,0)
print(outcome(e))
```

```
Player.one
```

[89]:
```
e = (Player.two,5)
try:
    print(outcome(e))
except Not_Terminal as nt:
    print(nt)
```

```
Etat non terminal
```

2.0.6 Q6

Ecrire la fonction `move(e:(Player,int))->[(Player,int)]` qui renvoie la liste des états voisins de l'état `e`, c.a.d les états qu'on peut joindre en un coup à partir de `e`.

[93]:
```
e = (Player.two,2)
etats = move(e)
for etat in etats:
    print("player:{},nb d'allumettes dans le tas = {}".format(player(etat).
 ↪name,quantity(etat)))
```

```
player:one,nb d'allumettes dans le tas = 1
player:one,nb d'allumettes dans le tas = 0
```

2.0.7 Q7

Ecrire la fonction `pred(e:(Player,int))->[(Player,int)]:` qui retourne la liste des états qui ont `e` comme voisin.

[95]:
```
e = (Player.one,init-2)
etats = pred(e)
for etat in etats:
    print("player:{},nb d'allumettes dans le tas = {}".format(player(etat).
 ↪name,quantity(etat)))
```

```
player:two,nb d'allumettes dans le tas = 16
```

2.0.8 Q8

Ecrire la fonction `maj(j:Player,a:[State],d:dict)->[State]` qui correspond à une mise à jour de l'attracteur.

- `a` est la liste des états qui ont été ajoutés *au tour précédent* à l'attracteur;
- `j` est le joueur dont on calcule l'attracteur;
- `d` est un dictionnaire dont les clés sont des états et les valeurs sont quelconques. Si l'état `e` est une clé de `d`, alors `e` est dans l'attracteur.

La fonction renvoie la liste des nouveaux états ajoutés à l'attracteur.

[97]:
```
e = (Player.two,0)
a = [e]
d = {e:True}
a=maj(Player.one,a,d)
for e in d :
    print(e)
print("-----------")
for e in a:
    print(e)
```

```
(<Player.two: 2>, 0)
(<Player.one: 1>, 1)
(<Player.one: 1>, 2)
(<Player.one: 1>, 3)
-----------
(<Player.one: 1>, 1)
(<Player.one: 1>, 2)
(<Player.one: 1>, 3)
```

2.0.9 Q9

Ecrire la fonction `attracteur(j)` qui prend en paramètre un joueur et retourne son l'attracteur : le dictionnaire dont les clés sont les états pour lesquels il existe une stratégie gagnante pour `j`.

[100]:
```
for e in attracteur(Player.two):
    print(e)
```

```
(<Player.one: 1>, 0)
(<Player.two: 2>, 1)
(<Player.two: 2>, 2)
```

```
(<Player.two: 2>, 3)
(<Player.one: 1>, 4)
(<Player.two: 2>, 5)
(<Player.two: 2>, 6)
(<Player.two: 2>, 7)
(<Player.one: 1>, 8)
(<Player.two: 2>, 9)
(<Player.two: 2>, 10)
(<Player.two: 2>, 11)
(<Player.one: 1>, 12)
(<Player.two: 2>, 13)
(<Player.two: 2>, 14)
(<Player.two: 2>, 15)
(<Player.one: 1>, 16)
```

2.0.10 Q10

Ecrire la fonction `minmax(j:Player,e:(Player,int))->int` qui renvoie +1 ou -1 selon l'algorithme du minmax vu en cours.

[108]:
```
minmax(Player.one,(Player.one,init))
```

[108]: 1

[109]:
```
minmax(Player.one,(Player.one,4))
```

[109]: -1

A suivre : minmax avec profondeur maximale et heuristique (qui évalue les chances de gagner selon le point de vue d'un joueur).

[ ]:
```

```