Projet : Jeux séquentiels et stratégies en python

Consignes

- **Par définition, le respect des consignes est exigée :-)**
- Pour toute question, n'hésitez pas à utiliser le channel mattermost DJ23-24.

- Ce projet est à réaliser seul ou en binôme.
  - ▶ Vous devez rendre un fichier zip contenant votre projet. Le fichier se nomme `nom1-nom2.zip` où `nom1` et `nom2` sont les noms des membres du binôme.
  - ▶ Chaque membre du binôme doit soumettre le même fichier zip.
- Le fichier zip doit contenir un fichier `projet.py` et un fichier `rapport.ipynb` (notebook jupyter).
  - ▶ Le fichier `projet.py` doit contenir les classes et les fonctions que vous avez implémentées.
  - ▶ Le fichier `rapport.ipynb` doit contenir une description de votre projet, les explications (si besoin) sur les classes et les fonctions que vous avez implémentées, des exemples d'utilisation, des statistiques sur les performances des différentes stratégies, etc.
- Une attention particulière doit être portée à la qualité du code (documentation, commentaires, noms des variables, etc.).
- Respectez le nom des classes et méthodes données dans l'énoncé. N'hésitez pas à créer classes, méthodes et fonctions qui vous semblent utiles.
- Vous remarquerez, en particulier, qu'aucune visualisation n'est spécifiée. Le sujet vous laisse toute liberté pour les modes de visualisations des jeux et des parties.

Énoncé du Projet

Soit un jeu séquentiel, à somme nulle et à information complète défini par :

- Deux joueurs nommés J_1 et J_2.
- Chaque joueur à un nombre fini de coups.
- Soit C la configuration courante du jeu : C détermine parfaitement l'état du jeu (et peut contenir, par exemple, le numéro du coup, l'historique des coups joués, quel joueur joue le prochain coup, l'état de l'échiquier... Il pourrait contenir également un temps restant pour les Blitz, etc.).
- On suppose qu'il existe $f_1(C) \in \mathbb{R}$, la fonction d'évaluation de J_1 pour la configuration C (plus $f_1(C)$ est grande, plus la configuration est favorable au joueur 1). Par définition, $f_2(C) = -f_1(C)$.
- On suppose que le jeu est fini, c'est-à-dire que toute partie se termine en un nombre fini de coups.

Exercice 1 – Représentation des jeux

Q 1.1 Classe abstraite `JeuSequentiel`

En python (et en C++ d'ailleurs), une classe abstraite est ce qui est le plus proche de la notion d'interface en Java : la déclaration des méthodes à implémenter mais sans codes associés. En python, cela se traduit par des méthodes qui ont pour code l'envoi d'une exception `NotImplementedError`.
Proposer une classe python abstraite, qui permette de décrire complètement (et le plus généralement possible) un jeu à somme nulle, séquentiel et à information complète. Par exemple, il sera nécessaire de choisir la structure de donnée représentant une configuration C pour chaque jeu. Toutefois, on peut spécifier la fonction qui donne le prochain joueur dans la configuration C, la fonction qui donne les coups possibles dans la configuration C, la fonction d'évaluation, etc.
Un extrait de cette classe abstraite est donné ci-dessous (vous pouvez l'améliorer, la modifier, la compléter, etc.) :

```python
class JeuSequentiel:
  """
  Represente un jeu sequentiel, a somme
    nulle, a information parfaite
  """
  def __init__(self):
    pass

  def joueurCourant(self, C):
    """
    Rend le joueur courant dans la
    configuration C
    """
    raise NotImplementedError

  def coupsPossibles(self, C):
    """
    Rend la liste des coups possibles dans
     la configuration C
    """
    raise NotImplementedError

  def f1(self, C):
    """
    Rend la valeur de l'evaluation de la
    configuration C pour le joueur 1
    """
    raise NotImplementedError

  def joueLeCoup(self, C, coup):
    """
    Rend la configuration obtenue apres
    que le joueur courant ait joue le coup
     dans la configuration C
    """
    raise NotImplementedError

  def estFini(self, C):
    """
    Rend True si la configuration C est
    une configuration finale
    """
    raise NotImplementedError
```

Q 1.2 Classe `Morpion`
De cette classe abstraite, on peut ensuite dériver des classes "concrètes" qui implémentent toutes ces méthodes.
Proposer une classe dérivée de cette classe qui décrit le jeu du morpion

```
class Morpion(JeuSequentiel):
  """
  Represente le jeu du morpion (3x3)
  """
  ...
```

Exercice 2 – Représentation des stratégies

Q 2.1 Classe abstraite `Strategie`
Proposer une classe abstraite `Strategie` qui permet de décrire une stratégie de jeu. Cette classe encapsule principalement une fonction `choisiProchainCoup` qui prend en paramètre une configuration C et retourne le coup choisi par la stratégie. En voici un extrait (libre à vous de modifier, ajouter, etc.) :

```
class Strategie:
  """
  Represente une strategie de jeu
  """
  def __init__(self,jeu:JeuSequentiel)):
    pass

  def choisirProchainCoup(self, C):
    """
    Choisit un coup parmi les coups possibles dans la configuration C
    """
    raise NotImplementedError
```

Q 2.2 Class `StrategieAleatoire`
Proposer une classe concrète, dérivée de `Strategie` représentant une stratégie aléatoire qui fonctionne pour tout `JeuSequentiel`.

```
class StrategieAleatoire(Strategie):
  """
  Represente une strategie de jeu aleatoire pour tout jeu sequentiel
  """
  def __init__(self,jeu:Jeu)):
    ...

  def choisirProchainCoup(self, C):
    """
    Choisit un coup aleatoire suivant une distribution uniforme sur tous les coups
  possibles dans la configuration C
    """
    ...
```

Q 2.3 Première simulation
Écrire une fonction qui exécute une partie de morpion entre deux joueurs jouant cette stratégie aléatoire.

Exercice 3 – Stratégie min-max

Dans le cadre d'un jeu séquentiel à somme nulle, le joueur J_1 cherche à maximiser $f_1(C)$, tandis que le joueur J_2 cherche à minimiser $f_1(C)$.

Supposons que le jeu soit dans la configuration C et que ce soit au tour de J_1 de jouer.

1. Si J_1 a n coups possibles dans la configuration C, notons $C_1, C_2, \ldots, C_n$ les configurations obtenues en jouant ces coups.
2. J_1 pourrait donc choisir le coup i' qui maximise $f_1(C_{i'})$.
3. Toutefois, dans chacune de ces configurations C_i, J_2 devra jouer et choisira un coup x_i^* qui minimisera $f_1(C_{i,x_i})$.
4. Finalement, J_1 devrait choisir le coup i^* qui maximise $f_1(C_{i^*,x_{i^*}^*})$.

L'item 2 propose une stratégie pour J_1 de niveau d'analyse 1, l'item 4 propose une stratégie pour J_1 de niveau d'analyse 2. Il est facile de généraliser et de proposer par récurrence une stratégie pour J_1 de niveau d'analyse $k > 0$.
On peut ainsi construire un arbre de jeu de profondeur k où chaque nœud est une configuration du jeu et chaque arête est un coup possible. Les nœuds de profondeur paire sont des nœuds de type J_1 et les nœuds de profondeur impaire sont des nœuds de type J_2. Les feuilles de l'arbre sont les configurations finales du jeu. Une branche de l'arbre est une séquence de coups qui mène à une configuration finale et représente donc les k possibles prochains coups d'une partie commennçant dans la configuration C.
Il existe forcément une valeur de k pour laquelle l'arbre de jeu est complet, c'est à dire que toutes les feuilles contiennent une configuration finale du jeu.

Q 3.1 Classe `StrategieMinMax`
Implémenter une stratégie MinMax de profondeur k pour tout `JeuSequentiel`.
Proposer une structure de données qui permette de représenter et de résoudre un arbre de jeu de profondeur k pour un jeu de type `JeuSequentiel`.

```
class StrategieMinMax(Strategie):
  """
  Represente un strategie utilisant un arbre min-max de profondeur k
  """
  def __init__(self, jeu:JeuSequentiel, k:int, ...)
    ...
```

Exercice 4 – Application au jeux des allumettes

Le jeu des allumettes est un jeu à somme nulle, séquentiel et à information complète. Il se joue à deux joueurs, sur une table avec un nombre (N) d'allumettes disposées en g groupes de m_i allumettes chacune ($N = \sum_{i=1}^{g} m_i$). Par exemple, $m_i = 5$ pour tout groupe, ou alors $m_i = i$, etc. Les joueurs jouent à tour de rôle et retirent un nombre d'allumettes quelconque d'un des groupes. Le joueur qui retire la dernière allumette a perdu.

Q 4.1 Classe `Allumettes`
Proposer une classe dérivée de `JeuSequentiel` qui décrive le jeu des allumettes. Attention à la fonction d'évaluation !

```
class Allumettes(JeuSequentiel):
  """
  Represente le jeu des allumettes pour $g$ groupe de $m$ allumettes
  """
  def __init__(self, g:int, m:int, ...)
    ...
```

Q 4.2 Stratégie optimale
Proposer une implémentation de la stratégie optimale proposée dans le TD9 (Jeu de Nim, fonction de Grundy) pour résoudre le jeu des allumettes.

```python
class StrategieAllumettes(Strategie):
  """
  Represente la strategie optimale pour le jeu des allumettes
  """
  def __init__(self, jeu:Allumettes)
    ...
```

Exercice 5 – Simulations

Vous avez donc maintenant une stratégie aléatoire, des stratégies MinMax de profondeur k et une stratégie optimale pour le jeu des allumettes. Vous pouvez donc facilement organiser des tournois et calculer des statistiques sur les performances des différentes stratégies (il n'y aura pas trop de surprises normalement, mais c'est toujours intéressant de vérifier).
Cette partie est libre. N'hésitez pas à calculer des statistiques sur les performances des différentes stratégies, à comparer les performances des stratégies MinMax pour différentes valeurs de k, à comparer les performances des stratégies MinMax avec la stratégie optimale, etc. en victoire, en temps, en durée de partie, etc.