Fiche TP01-A

marc-michel.corsini@u-bordeaux.fr

31 janvier 2015

Ce premier TP consiste à construire le jeu de Doo à partir de la classe abstraite `Game` du fichier `abstract.py`. Pour vous aider, une version du jeu des allumettes est fournie dans `allumettes.py`. Vous ne **devez pas** modifier les fichiers fournis à moins que j'en ai fait une demande **explicite**.
Un certain nombre de contraintes sont imposées afin de pouvoir passer dans les moulinettes génériques que j'utilise.

1 Description générale

Un jeu est défini par un **état** ou **configuration** ; cet état va évoluer au cours de la partie, il pourra donc être intéressant de garder l'historique des configurations, afin de permettre de revenir dans une configuration antérieure, ou de savoir si cette situation a déjà été rencontrée par le passé. Un jeu doit permettre d'accéder aux règles du jeu, il doit pouvoir indiquer si une configuration est terminale ou non. Il doit pouvoir, étant donné le joueur, trouver le joueur adverse. Il doit être capable de calculer la liste des coups possibles étant donnés la configuration courante et un joueur. Il faut enfin être capable de passer d'une configuration à une autre, étant donnés la configuration courante, le joueur qui a le trait et le coup à effectuer. Il doit être capable de dire si, étant donné l'état courant, étant connu un joueur, ce joueur a gagné (ou perdu). De plus, le jeu doit permettre de fournir une évaluation numérique de la configuration courante - cette évaluation servira lorsqu'il faudra mettre en place des joueurs « intelligents ».

2 Codage

Le développement sera fait en Python3.xx avec un style de programmation orientée objets. Vous **devez** respecter les signatures et le nom des méthodes ou fonctions à mettre en œuvre, des programmes python vous seront fournis afin de valider les premières étapes de votre projet. La validation du code fait partie de l'évaluation finale de votre projet.
Attention ne confondez pas les entrées/sorties (imposées) avec le codage interne que vous utilisez.

3 Doo : les besoins

Dans la fiche `presentation.pdf` a été fourni un descriptif du jeu. Il y a deux joueurs qui sont opposés, un tablier de 12 cases, 3 types de pierres (noires, blanches et Roi). Une partie se déroule en plusieurs manches, chaque manche est constituée de deux phases. Il faut comptabiliser des points de manches.

3.1 Variables globales

Pour pouvoir effectuer des tests de validation pour tous, j'ai besoin de connaitre certaines valeurs de codage. C'est pourquoi en début de fichier `tp01.py` vous devrez spécifier le code que vous utilisez pour désigner : les deux joueurs, les pierres blanches, les pierres noires et le roi. Ces valeurs ne sont pas obligatoirement identiques à celles qui seront affichées à l'écran, mais elles sont utilisées (par vous) pour coder l'information - et par moi, dans les tests pour vérifier que votre code correspond aux besoins. Par exemple, vous pouvez très bien coder une case vide avec la valeur -1 tandis qu'à l'écran vous affichez un espace (ou un point).

3.2 Structures exportées

Une configuration ou état, représente toute l'information nécessaire pour pouvoir décider de la prochaine action à entreprendre. Pour le jeu de Doo une configuration sera une paire dont le premier élément sera le tablier, le second sera le numéro du (demi) tour. Toujours pour des raisons de flexibilité des codes de validation :

1. Le tablier sera, dans une configuration, représenté par une liste (ou un tuple) uni-dimensionnel ;
2. La case A1 sera la première position de la liste, la case C4 sera la dernière position de la liste, les positions seront donc successivement A1, B1, C1, A2, B2, C2, ... A4, B4, C4.
3. Un coup sera représenté de différentes façons en fonction de la phase de jeu et de sa nature :
   - Pendant la phase de pose, le coup aura le format (type_pion,position) où type_pion sera à valeur dans l'ensemble { NOIRS, ROI } et position sera un nombre appartenant à 0 ... 11. Ou il sera de la forme paire_pos, où paire_pos sera une liste ou un tuple python (p1, p2) avec p1 > p2
   - Pendant la phase de duel, le coup aura le format
     (a) (position1,position2), lorsqu'il s'agit d'un déplacement sans capture, position1 et position2 sont à valeur dans 0..11
     (b) (position,liste_positions) où position est un nombre appartenant à 0 ... 11, liste_positions sera une liste (ou un tuple) python des positions occupées successivement par le pion pendant sa prise.

Ainsi en reprenant l'exemple de la fiche directive pour le défenseur on aura :

1. A1 - C1 donnera (0,[2])
2. A1 - A2 donnera (0,3)
3. A4 - C4.C2 donnera (9,[11,5])

3.3 AdT pour les signatures

Avant de vous fournir les signatures, et les axiomes des méthodes à réaliser, quelques notations (qui découlent des « Structures exportées »)

- **Pions** désigne l'ensemble { BLANCS, NOIRS, ROI, VIDE } ; de fait VIDE n'est pas un vrai pion c'est plutôt une absence de vrai pion. On notera « tpion » un élément de { BLANCS, NOIRS, ROI }
- un « tablier » est donc un élément de **Pions**^{12} et **Tabliers** désigne l'ensemble des tabliers possibles
- **Etats** est l'ensemble des états ou des configurations, un élément de l'ensemble sera noté « cfg »
- **String** l'ensemble des chaînes de caractères
- **Joueurs** désigne l'ensemble { J_ATT, J_DEF }
- **Coups** désigne l'ensemble des coups jouables (autorisés)
- **Positions** désigne l'ensemble des positions du tablier
- **Libres** désigne l'ensemble des positions où un joueur peut poser/déplacer un de ses pions

4 Signatures, axiomes

Le jeu de Doo est complètement décrit par sa configuration, son état. La variable `self` sera donc une configuration, on note « cfg_0 » l'état créé par l'initialisation.

1. `__init__` : $\emptyset \rightarrow \emptyset$
   axiome : crée une configuration initiale, $\texttt{cfg}_0$, qui est [VIDE, VIDE, VIDE, VIDE, BLANCS, VIDE, VIDE, VIDE, VIDE, VIDE, VIDE, VIDE], 1

2. `__str__` : **Etats** → **String**
   La chaîne de caractères correspondant à la configuration courante.
   axiomes : aucun

3. `configuration` est une abbréviation pour deux fonctions

   - `get_configuration` : **Etats** → **Etats**
     renvoie la configuration courante
   - `set_configuration` : **Etats** → ∅
     modifie la configuration courante

   axiomes :

   ```
   #1 get_configuration
   p = Doo()
   p.configuration == cfg0
   #2 set_configuration
   p.configuration = cfg
   p.configuration == cfg
   ```

4. `regles` : **Etats** → **String**
   renvoie une chaîne de caractères correspondant à la règle du jeu
   axiomes :

   ```
   p.regles() == Doo.regles()
   ```

5. `adversaire` : **Etats** x **Joueurs** → **Joueurs**
   axiomes :

   ```
   p.adversaire( J_ATT ) == J_DEF
   p.adversaire( J_DEF ) == J_ATT
   ```

6. `listeCoups` : **Etats** x **Joueurs** → **Coups**^{k}
   Coups^0 désigne ∅, pour $k > 0$ on obtient un k-uplet d'éléments de **Coups** .
   axiomes :

   ```
   p = Doo()
   p.listeCoups( J_DEF ) == [] # correspond à ∅
   p.listeCoups( J_ATT ) ⊂ { NOIRS, ROI } x Libres
   ```

   - Si joueur n'a pas le trait : p.listeCoups(joueur) == [ ]
   - Si J_ATT a gagné : p.listeCoups(joueur) == [ ]

7. `gagnant` : **Etats** x **Joueurs** → 𝔹
   axiomes : renvoie vrai si et seulement si le joueur a gagné

8. `perdant` : **Etats** x **Joueurs** → 𝔹
   axiomes : renvoie vrai si et seulement si le joueur a perdu

9. `finPartie` : **Etats** x **Joueurs** → 𝔹
   axiomes : renvoie vrai si et seulement si la *manche* est terminée
   finPartie(joueur) == finPartie(adversaire(joueur))

10. `joue` : **Etats** x **Joueurs** x **Coups** → **Etats**
    axiomes :

    (a) Ne modifie pas l'état courant du jeu ;

(b) La configuration renvoyée correspond à l'effet du coup sur la configuration courante.

(c) soit cfg[1] le compteur de (demi-) tours d'une configuration alors on a la propriété

```
p.configuration[1] + 1 == p.joue(joueur,coup)[1]
```

11. `evaluation` : **Etats** x **Joueurs** → $\mathbb{R}$
    C'est une évaluation numérique estimant si l'état courant est favorable ou non au joueur.
    axiomes :

    (a) `min` évaluation si p.perdant(joueur)

    (b) `max` évaluation si p.gagnant(joueur)

    (c) `min` == -1 * `max`

    (d) `min` ≤ `p.evaluation(joueur)` ≤ `max`

5 Réalisation

Écrivez votre code en pensant à l'objectif : réussir la validation qui s'occupe de contrôler que votre code respecte les signatures et les axiomes.
Vous ne devez pas chercher à vérifier si les paramètres sont ceux attendus, préoccupez vous des cas où l'information reçue respecte la signature et fournissez le résultat attendu.
N'écrivez pas tout en une seule étape, fractionnez votre travail et validez chaque partie finalisée avant de passer à la suivante.

1. `tp01_skeleton.py` va vous servir à créer votre fichier, je vous encourage à le nommer `tp01.py`. Dans tous les cas **INTERDICTION** d'utiliser un nom de fichier avec des accents ou des espaces. Utilisez uniquement les 26 lettres de l'aphabet et les 10 chiffres.

2. `validation01.py` est le fichier permettant de vérifier que tout va bien, pour pouvoir l'utiliser il faudra modifier les lignes 11 et 12 en accord avec votre implémentation. Puis lancer son exécution (via votre EDI[1], ou par ligne de commande).

Pour chaque méthode `meth` à implémenter, il y a une fonction test associée `test_meth`, il est donc possible en mode interactif de passer un, ou plusieurs, test.

```
mmc@vostro-mmc: Code python3 -i validation01.py
TESTS: .........
----------------------------------------------------------------------
tests reussis: 24/31 (77.42%)
>>> test_adversaire()
'..'
>>> test_joue()
'E'
>>>
```

Si tout se passe bien, vous obtenez une séquence de points, en cas d'erreur un E est affiché.

[1] « Environnement de Développement Intégré » en grand breton on parle d'*Integrated Development Environment (IDE)*