IFT 615 : Devoir 1
Travail individuel

Remise : 18 mai 2015, 9h00 (**au plus tard**)

Ce devoir comporte 3 questions de programmation. Vous trouverez tous les fichiers nécessaires pour ce devoir ici : `http://marccote.github.io/courses/ift615/devoir_1/devoir_1.zip`.

Veuillez soumettre vos solutions à l'aide de l'outil **turnin** :

```
turnin -c ift615 -p devoir_1 solution_taquin.py solution_tictactoe.py solution_connect4.py
```

1. [**5 points**] Programmez l'algorithme A* pour résoudre le jeu du taquin à 3×3 cases (*8-puzzle* : `http://fr.wikipedia.org/wiki/Taquin`).

   Le programme doit être dans le langage Python. Plus spécifiquement, vous devez remettre un fichier `solution_taquin.py` contenant une fonction nommée `joueur_taquin` prenant 4 arguments (dans l'ordre) :

   - `etat_depart` : Objet de la classe `TaquinEtat` indiquant l'état initial du jeu.
   - `fct_estEtatFinal` : Fonction qui prend en entrée un objet de la classe `TaquinEtat` et qui retourne `True` si l'état passé en paramètre est l'état final.
   - `fct_transitions` : Fonction qui prend en entrée un objet de la classe `TaquinEtat` et qui retourne la liste des états voisins (objets `TaquinEtat`) pour l'état donné. **On suppose que le coût de chaque transition est de 1**.
   - `fct_heuristique` : C'est la fonction heuristique à utiliser. Cette fonction prend en entrée un objet `TaquinEtat` et retourne l'heuristique pour cet état.

   Votre fonction `joueur_taquin` doit retourner la solution au jeu, c'est-à-dire la liste des objets `TaquinEtat` correspondant à la solution trouvée par A* (de l'état de départ vers l'état final inclusivement, dans l'ordre).

   Un fichier initial `solution_taquin.py` vous est fourni. Il contient entre autres la définition d'une classe `AEtoileTuple`, qui joue le rôle des triplets $(n, f(n), parent)$ utilisés par A* dans ses listes *open* et *closed*. Cette classe a les méthodes suivantes :

   - `__init__(self, etat, f, parent=None)` : Constructeur ayant comme argument `etat` (objet `TaquinEtat`) qui correspond à un état n, `f` (nombre à virgule flottante) qui correspond à la valeur de la fonction d'évaluation $f(n)$ pour cet état, ainsi que `parent` (objet `AEtoileTuple`) correspondant au parent de l'état n dans la recherche (avec `None` comme valeur par défaut, appropriée pour la racine de l'arbre de recherche). Une fois un objet `AEtoileTuple` construit, ces arguments sont accessibles comme des champs de l'objet (`objet.etat`, `objet.f` et `objet.parent`).
   - `__eq__(self, autre)` : Retourne `True` si l'objet `autre` (aussi de la classe `AEtoileTuple`) correspond au même état du jeu. Cette méthode est invoquée lors d'un test d'égalité (`==`) entre deux objets.
     Ceci permet d'avoir le comportement suivant : si `atuple` est un objet `AEtoileTuple` et que `open` est une liste d'objets `AEtoileTuple`, alors le test `atuple in open` retournera `True` si `open` contient un objet `AEtoileTuple` ayant le même état que `atuple`.

   À noter que `parent` doit pointer vers un objet `AEtoileTuple` et non vers un état (objet `TaquinEtat`), afin que vous puissiez remonter vers la racine de l'arbre de recherche et retourner la solution finale, à la fin de A*.

Le script Python `taquin.py` importera la fonction `joueur_taquin` contenue dans `solution_taquin.py` (spécifié via le paramètre `-joueur`) et l'utilisera afin de résoudre le jeu du Taquin à partir d'une configuration initiale fixée.

Voici comment utiliser ce script :

```
Usage: python taquin.py [-joueur JOUEUR] [-no_partie INT] [-valider FICHIER]

où -joueur    : "humain" ou le fichier contenant votre solution.
                Défaut = "solution_taquin.py"
   -no_partie : un entier quelconque
                Défaut = partie utilisée pour générer "taquin_validation.pkl"
   -valider   : un fichier permettant de valider votre joueur pour un jeu donné.
                Défaut = "taquin_validation.pkl"
```

Par défaut, le jeu est toujours initialisé à la même configuration et la solution de votre agent est comparée à la solution attendue, contenue dans le fichier `taquin_validation.pkl` (spécifié via le paramètre `-valider`). Le fichier `taquin_validation.txt` contient également une impression de la solution attendue. Si vous souhaitez varier la configuration initiale afin de tester votre agent sur d'autres configurations, changez le numéro de partie via le paramètre `-no_partie` (un entier positif, qui sera utilisé par le générateur aléatoire de configuration comme germe ou *seed*). Notez que certaines configurations pourraient nécessiter plus de temps pour être résolues. Si vous désirez voir comment ce comporte votre agent, utilisez l'option `-v` pour faire afficher la grille après chaque coup.

Conseils : dans le cours, on a toujours considéré que la liste *open* était maintenue ordonnée. En fait, il peut être plus simple de seulement s'assurer que le premier élément soit celui associé à la valeur de fonction d'évaluation la plus petite. Vous pouvez même ne pas ordonner la liste du tout et trouver le minimum à chaque fois que vous sortez l'état courant de *open*. À vous de décider, pourvu que votre choix donne une implémentation qui ne soit pas trop lente. Finalement, si vous souhaitez analyser l'efficacité de votre code, il suffit d'ajouter `-m cProfile` lors de l'exécution (par exemple `python -m cProfile taquin.py`).

2. [**5 points**] Programmez un joueur optimal de *tic-tac-toe* en implémentant l'algorithme de recherche alpha-bêta. La recherche alpha-bêta devra donc se faire jusqu'à la profondeur maximale.

   Le programme doit être dans le langage Python. Plus spécifiquement, vous devez remettre un fichier nommé `solution_tictactoe.py` contenant une fonction nommée `joueur_tictactoe` prenant 4 arguments (dans l'ordre) :

   — `etat` : Objet `TicTacToeEtat` représentant l'état actuel du jeu.
   — `fct_but` : Une fonction d'utilité prenant un état en argument et qui retourne l'utilité associée à l'état (grand nombre positif si le joueur `'X'` gagne, grand nombre négatif s'il perd, 0 si c'est une partie nulle). Si l'état ne correspond pas à une partie terminée, `fct_but` retourne `None`.
   — `fct_transitions` : Une fonction acceptant un état comme argument et donnant en sortie un dictionnaire qui associe chaque action possible (clé du dictionnaire) à l'état qui en résulte (valeur du dictionnaire).
   — `str_joueur` : Le rôle joué par ce joueur, c'est-à-dire `'X'` ou `'O'` (sous forme d'une chaîne de caractères).

   La fonction `joueur_tictactoe` doit retourner l'action prise par votre joueur. Cette action doit donc être une clé valide du dictionnaire retourné par `fct_transitions`.

   Le script Python `tictactoe.py` importera la fonction `joueur_tictactoe` contenue dans `solution_tictactoe.py` (qui doit être dans le même répertoire) et l'utilisera pour simuler le joueur `'X'`. Pour tester votre code, vous pouvez jouer vous-même le joueur `'O'`, invoquer un joueur aléatoire ou utiliser la même fonction `joueur_tictactoe`. Voici comment utiliser `tictactoe.py` :

```
Usage: python tictactoe.py [-joueur1 JOUEUR] [-joueur2 JOUEUR]

où -joueur1 : "humain", "aléatoire" ou le fichier contenant votre solution.
              Défaut = "solution_tictactoe.py"
   -joueur2 : "humain", "aléatoire" ou le fichier contenant votre solution.
              Défaut = "solution_tictactoe.py"
```

   Si vous désirez voir comment ce comporte votre agent, utilisez l'option `-v` pour faire afficher la grille après chaque coup.

   La correction de votre code se fera de façon automatique, en vérifiant automatiquement l'ensemble des états visités par votre recherche alpha-bêta. Puisque l'ensemble des états visités dépend de l'ordre dans lequel les états successeurs sont générés, on vous demande de toujours utiliser l'ordre suggéré par Python en itérant sur les items du dictionnaire :

```
for action,nouvel_etat in fct_transitions(etat).items():
    # Votre code
    ...
```

3. **[BONUS]** Implémentez également un joueur pour le jeu Connect 4 (Puissance 4 : `http://fr.wikipedia.org/wiki/Puissance_4`). Normalement, votre code pour le tic-tac-toe devrait également fonctionner pour Connect 4. Par contre, une recherche jusqu'à la profondeur maximale est trop lente dans le cas de Connect 4. Ainsi, vous devrez l'ajuster afin de limiter la profondeur de la recherche et développer une heuristique pour compenser. Plus votre heuristique sera bonne, plus les chances de gagner de votre joueur seront bonnes.

   Pour ce faire, vous devez remettre un fichier nommé `solution_connect4.py` contenant une fonction `joueur_connect4`. Cette fonction doit prendre les mêmes arguments que `joueur_tictactoe`, en plus d'un cinquième argument :

   — `int_tempsMaximal` : le temps maximal, en secondes, accordé au joueur pour retourner une action.

   Si le joueur prend plus qu'une fois et demi (150%) du temps maximal accordé pour choisir une action, il sera éliminé et perdra par défaut. S'il prend moins que 150% du temps, mais plus de 100%, le temps excédant pris par le joueur sera alors retranché au temps accordé à son tour suivant.

   Pour tenir compte du temps écoulé, vous pouvez utiliser le module standard `time` et sa fonction `time()` qui retourne le temps (en secondes) écoulé depuis le 1er janvier 1970 (pour les systèmes Unix). Plus de détails sur le module `time` sont disponibles ici : `http://docs.python.org/library/time.html`.

   Pour développer votre fonction d'évaluation heuristique, vous pourrez évaluer la qualité d'un état à partir du champ `etat.tableau` de celui-ci. Ce champ est un tableau Numpy de caractères. Il contient l'information sur l'occupation de chaque case du jeu, à toutes les positions possibles : chaque entrée vaut donc `'X'`, `'O'` ou `' '` (pour une case non occupée). Par exemple, l'état du jeu

```
| | | | | | | |
| | | | | | | |
| |X| | | | | |
| |X| |O| | | |
| |X| |O| | | |
| |X| |O| | |O|X|
-----------------
 0 1 2 3 4 5 6 7
```

   correspond à la valeur de `etat.tableau` suivante :

```
[[' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' 'X' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' 'X' ' ' ' 'O' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' 'X' ' ' ' 'O' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']
 [' ' 'X' ' ' ' 'O' ' ' ' ' ' ' 'O' 'X']]
```

   L'évaluation de votre solution se fera sous la forme d'un tournoi, où tous les joueurs soumis auront à joueur un contre l'autre. **Deux points boni** seront accordés aux 5 meilleurs joueurs du tournoi. **Un point boni** ira aux 5 suivants. Par contre, pour obtenir ces points, les joueurs devront également avoir fait mieux qu'un joueur (gardé secret) utilisant une heuristique très simple.