A Game of Tron

Partie I

Introduction

Tron est un film de science-fiction américain réalisé par Steven Lisberger, sorti en 1982. Une suite, Tron : L'Héritage, a été réalisée en 2010 par les studios Disney. Dans les deux films, des personnes réelles (Kevin Flynn dans le premier opus, son fils Sam dans le deuxième) sont envoyées dans un ordinateur et doivent affronter des programmes dans diverses épreuves, notamment en se déplaçant avec une moto qui ne peut s'arrêter sur une carte bidimensionnelle parsemée d'obstacles. Seulement, toutes les cases visitées précédemment par les deux joueurs se transforment elles aussi automatiquement en obstacle et il n'est plus possible d'y pénétrer.

Vous l'avez compris, il s'agit de survivre le plus longtemps possible sans exploser sa moto sur un mur. Ceci-dit, contrairement aux jeux que l'on propose aux humains et qui testent leurs réflexes et leurs nerfs, ce TP ne s'appuie que sur la réflexion pour choisir le chemin qui maximisera la durée de vie. Force est de constater que si le cerveau humain est assez facilement capable de s'adapter au comportement de l'adversaire, il est bien plus difficile de programmer une intelligence artificielle qui ferait les mêmes choix optimaux qu'un humain pourrait faire. Tel est le défi posé par le tournoi à l'adresse `http://troncontest.kleber.free.fr/`.

Partie II

Mise en place du jeu

Tout est indiqué sur le site web précédemment cité. Lisez les docs, chargez le « starter pack » et testez le comportement par défaut pour voir si cela fonctionne (ne pas hésiter à demander en cas de souci).

Partie III

Utiliser les outils disponibles

Pour écrire votre robot, vous aurez à modifier dans le fichier `MyTronBot.py` la fonction (déjà écrite en partie) `mouvement_choisi(carte)` qui reçoit en argument la carte sur laquelle va évoluer votre robot et doit renvoyer une direction parmi celles du quadruplet `DIRECTIONS = (NORD,EST,SUD,OUEST)`. La carte que l'on vous fournit dispose déjà de méthodes intéressantes:

`carte[x,y]:` permet d'obtenir la nature du terrain à la position `(x,y)` de la carte (le point de coordonnées `(0,0)` est en haut à gauche, alors que celui de coordonnées `(carte.largeur-1,carte.hauteur-1)` est tout en bas à droite). Ce terrain peut-être de quatre natures différentes: `MUR`, `VIDE` (c'est-à-dire que l'on peut essayer de s'y déplacer), `MOI` (votre position, mais équivalent à un mur pour toutes les fonctions utiles) et `LUI` (la position de l'adversaire, mais équivalent à un mur pour toutes les fonctions utiles). Par commodité, tout point hors de la carte est aussi considéré comme un mur.

`carte.largeur:` renvoie la largeur de la carte (exemple: `x in range(carte.largeur)`).

`carte.hauteur:` renvoie la hauteur de la carte (exemple: `y in range(carte.hauteur)`).

`carte.ma_position():` renvoie un doublet correspondant aux coordonnées de votre robot sur la carte.

`carte.position_adverse():` renvoie un doublet correspondant aux coordonnées du robot adverse.

`carte.traversable(position):` renvoie `True` si `carte[x,y] == VIDE`, `False` sinon où `position = (x,y)` est le doublet des coordonnées à regarder.

`carte.aller_vers(direction[,origine])`: indique les coordonnées de la case située dans la direction `direction` (à choisir parmi l'ensemble `DIRECTIONS = (NORD,EST,SUD,OUEST)`) par rapport à l'origine fournie (par défaut, la position de votre robot).

`carte.adjacents(position)`: donne *tous* les carrés adjacents à celui de coordonnées `position`.

`carte.mouvements_possibles()`: renvoie une liste contenant les directions (prises parmi les éléments du quadruplet `DIRECTIONS`) où il est possible d'aller. Renvoie la direction d'un (vrai) mur si jamais votre robot est entouré de toutes parts.

Outre ces méthodes, il peut être intéressant de définir une distance sur votre carte, par exemple la distance dite « Manhattan » qui indique le nombre de tours nécessaires pour que le robot situé en un point `p1` puisse atteindre le point `p2`:

```
1  def distance(p1,p2):
2      '''Calcul de la distance "Manhattan" entre deux points p1 et p2.'''
3      x1,y1 = p1
4      x2,y2 = p2
5      return abs(x2-x1) + abs(y2-y1)
```

Bien sûr, vous êtes libre de modifier le fichier `MyTronBot.py` pour rajouter autant de fonctions que vous le désirez, tant qu'elles peuvent vous être utiles pour faire le choix de la direction optimale à prendre.

Partie IV

Faire ses preuves

Avant de se lancer dans la conception d'algorithmes compliqués pour gagner à coup sûr tous les jeux de Tron, vous allez commencer par certains comportements simples:

L'escargot: S'amuse à toujours tourner dans le même sens (disons la droite) chaque fois que c'est possible. Vous vous doutez que ce n'est pas forcément optimal, mais au moins, ce n'est pas trop dur à coder (même si ce n'est pas totalement trivial).

L'amoureux des murs: Celui-ci aime longer les murs: l'idée est de compter les murs adjacents aux cases disponibles pour votre mouvement. Attention tout de même: s'il y a 4 murs adjacents à votre prochaine position, vous êtes bientôt mort...

Le chasseur: Le chasseur essaie toujours de se rapprocher de sa proie. Il faut donc calculer, à partir de chaque prochaine position accessible, celle qui est la plus proche de la position actuelle de l'adversaire. Attention, il a tendance à déclencher les matchs nuls plus souvent que les autres.

Le fuyard: Dans le même état d'esprit, le fuyard va toujours préférer la case qui est la plus éloignée de l'adversaire.

Pour ce faire, vous pouvez copier le fichier `MyTronBot.py` en, par exemple, `escargot.py`. Ensuite, il faut implémenter le comportement de l'escargot et pour le tester, il suffit de modifier le fichier `lance_test.py` en, par exemple,

```
1  import os
2  p1 = 'python escargot.py'
3  p2 = 'java -jar example_bots/Chaser.jar'
4  cmd= "java -jar ./engine/Tron.jar maps/duel-small.txt '%s' '%s'" % (p1,p2)
5  print('Execution de %s' % cmd)
6  os.system(cmd)
```