Fiche TP02-A

marc-michel.corsini@u-bordeaux.fr

21 mars 2016

1 Présentation générale

On va s'intéresser à la recherche d'aspirateurs ayant un comportement optimal vis à vis d'une certaine fonction d'évaluation. Mais avant de chercher l'optimalité, il va falloir mettre en place des aspirateurs qui savent agir en fonction d'un code génétique, c'est l'objectif de ce TP02-A, le TP02-B portera quant à lui sur la génération des codes génétiques.

1.1 Aspirateurs

Les aspirateurs vont disposer d'une variable « énergie » à valeur dans 0..100. Chaque action va avoir un coût (positif ou négatif) en énergie. Au départ (de chaque simulation), l'énergie sera à sa valeur maximale (100). L'objectif de ces aspirateurs sera multiple :

1. Nettoyer le plus de cases possibles au cours de la simulation. Chaque case nettoyée rapporte 10 points ;
2. Maximiser la quantité d'énergie en fin de simulation. Si au cours de la simulation l'énergie passe à 0 (ou moins), l'aspirateur tombe en panne (la variable **vivant** passe à `False`, la simulation s'arrête), la pénalité sera alors de **-100**.
   (a) si $1 \leq \textit{energie} < 25$ alors l'aspirateur marquera $\frac{1}{2} \times$ `energie`
   (b) si $25 \leq \textit{energie} < 50$ alors l'aspirateur marquera $\frac{2}{3} \times$ `energie`
   (c) si $50 \leq \textit{energie} < 75$ alors l'aspirateur marquera $\frac{3}{4} \times$ `energie`
   (d) si $75 \leq \textit{energie} \leq 100$ alors l'aspirateur marquera $\frac{1}{1} \times$ `energie`

getEvaluation() sera définie comme

```
score = (pieces_nettoyees / pieces_sales_au_depart) * 10
if not self.vivant:
    score -= 100
elif energie < 25 : score += 1/2 * energie / 100
elif energie < 50 : score += 2/3 * energie / 100
elif energie < 75 : score += 3/4 * energie / 100
else: score += energie / 100
```

1.1.1 Aspirateurs sans capteur

Ils évoluent dans un monde ligne $1 \times n$ ne contenant que deux sortes d'objets : 0 pour 'propre', 1 pour 'sale' ; ils disposent de 4 actions :

- Gauche, qui fait dépenser 1 point d'énergie ;
- Droite, qui fait dépenser 1 point d'énergie ;
- Aspirer, qui fait dépenser 5 point d'énergie ;
- Repos, qui fait gagner 3 points d'énergie.

1.1.2 Aspirateurs avec capteurs

Ils évoluent dans un monde ligne $1 \times n$ ne contenant que trois sortes d'objets : 0 pour 'propre', 1 pour 'sale', 2 pour 'prise électrique'. Pour tout $n > 3$, il y a **exactement** 3 prises dans le monde. On s'intéressera à 3 aspirateurs (en fonction de leurs capteurs) :

- [6,8] ;
- [8,2] ;
- [6,8,2].

Rq l'ordre des capteurs a une importance pour le codage/décodage des informations dans le code génétique.
Ces aspirateurs ont aussi les 4 mêmes actions mais les coûts sont légèrement différents pour l'action `Repos`

- Gauche, qui fait dépenser 1 point d'énergie ;
- Droite, qui fait dépenser 1 point d'énergie ;
- Aspirer, qui fait dépenser 5 point d'énergie ;
- Repos, qui fait gagner 20 points d'énergie si l'aspirateur est là où il y a une prise, 0 sinon.

1.2 Monde

Plusieurs paramètres sont fixés indépendamment de la nature des aspirateurs :

- mondes lignes $1 \times n$, avec $n \leq 15$.
- l'évaluation externe de l'aspirateur en situation **perfGlobale** va être définie comme :

$$\frac{\text{agent.getEvaluation()}}{\text{optimumTheorique}} - \text{nombre de fois où l'agent a choisi } \textbf{Repos} + n$$

- l'**optimumTheorique** est la valeur maximale de la fonction `getEvaluation`() pour la simulation courante
- le monde ne se préoccupe pas de savoir si l'aspirateur est un aspirateur aléatoire, avec base de connaissances ou génétique. Il se contente juste de fournir à l'aspirateur les informations dont celui-ci a besoin.

1.2.1 Remarques

Quelques petites précisions :

- La limitation $n = 15$ n'est pas réelle, mais permet de fixer les idées pour un travail **minimal** ;
- La limitation sur les objets a pour but de garantir le bon fonctionnement de votre code dans ces situations spécifiques, libre à vous d'ajouter d'autres contraintes pour expérimenter de nouveaux comportements ;
- Il n'y a pas de nombre maximum d'itérations pour la simulation – contrairement à la présentation faite **mardi 8 mars**. Les aspirateurs devront continuer à fonctionner pour des mondes de taille supérieure ;
- Il est tout à fait possible, une fois le TP réalisé pour les mondes à 1 ligne d'étendre au cas général i.e. monde $p \times n$ en augmentant le nombre de prises de manière raisonnable : **maximum** $20\% p \times n$, **minimum** 1 si $p \times n \geq 5$), en les plaçant à des endroits spécifiques. D'introduire de nouvelles actions de déplacement avec leur coût énergétique, de modifier les coûts, ...

1.3 Code génétique

Un « chromosome » peut être vu comme un programme, les composants du chromosome (les « gènes ») sont dans ce cas là les instructions du programme. La position du gène exprime la condition pour effectuer l'instruction, la valeur du gène étant l'instruction à effectuer si la condition est remplie.
Dans le cas d'un aspirateur sans capteur, la position du gène n'est que la position de l'instruction dans le programme. Pour un aspirateur avec capteur, la position du gène correspond à une valeur particulière de ce que peut percevoir l'aspirateur via ses capteurs.
Pour qu'un aspirateur lisant un code génétique puisse effectuer le programme demandé il a besoin de connaître le langage de programmation ainsi que la taille d'une instruction – exactement comme un ordinateur 32 bits sait qu'une instruction tient sur 32 bits et qu'un bit peut avoir les valeurs 0 ou 1 ; un ordinateur 32 bits ne sait pas comprendre le code utilisé par un programme 64 bits.

1.3.1 Outils spécifiques

Dans le fichier `briques.py`, la classe `ProgramGenetic` va permettre de manipuler simplement les informations le constructeur nécessite 4 paramètres :

1. La taille d'un gène, c'est le nombre de caractères de l'alphabet nécessaire pour coder une information ;
2. Le nombre de gènes présent dans le chromosome (le nombre d'instructions du programme) ;
3. L'alphabet (une liste de caractères) permettant de définir le langage ;
4. Un dictionnaire qui fait le lien entre le code génétique et sa signification (ici la signification est une action possible pour l'aspirateur).

Cette classe dispose d'un attribut en lecture/écriture **program** qui donne le code génétique, d'un attribut **actions** qui renvoie l'ensemble des actions codées et **decoder** qui prend en paramètre un index et renvoie l'action à cette position.
Attention La commande **len**`(x)` renvoie le nombre de gènes de x et non la longueur de la chaine de caractères obtenue par **len**`(x.program)`.
Toujours dans le fichier `briques.py`, la classe `GeneratePercept` va permettre d'associer à une perception particulière un entier. Le constructeur de la classe nécessite 2 paramètres :

1. La liste des capteurs à la disposition de l'aspirateur
2. Le dictionnaire des objets du monde

Ce code ne fait aucun controle de validité, il s'agit juste d'un petit outil pour faciliter votre travail. Une fois créée l'instance (appelons la `gp`), vous pouvez faire :

- `print(gp)` : affiche un résumé des informations
- `gp.howMany` : renvoie le nombre de cas possibles
- `gp.producer()` : renvoie un générateur des différents percepts, vous pouvez par exemple faire

```
compteur = 0
for x in gp.producer():
    if -1 in x: compteur += 1
print("{}% de cas avec bord".format(100 * compteur/gp.howMany))
```

- `gp.find( [-1,2] )` Permet de demander à quel index correspond la perception [-1,2], si par exemple vous avez opté pour la liste de capteurs [2,8]

2 Travail à effectuer

On va définir une classe `Aspirateur_PG` dérivée de `Aspirateur` et une classe `Monde_AG` dérivée de `Monde`. La construction va être assez similaire à celle du TP01. Le fichier `synopsis_02a.py` contient les éléments à développer.

2.1 Aspirateur_PG

Il faudra définir ou redéfinir :

- Le constructeur, prenant deux paramètres

    1. `prog` : une instance de `ProgramGenetic`. Par défaut prog est initialisé à `None` Dans ce cas, il faudra créer un ProgramGenetic basé sur l'alphabet ['A', 'G', 'D', 'R'] correspondant aux initiales des actions Aspirer, Gauche, Droite, Repos. Ce code aura une taille de gène de 1, et possèdera 8 gènes si la liste `lCap` est vide. Sinon, il faudra construire un ProgramGenetic compatible avec `lCap` et `objetsStatiques` – `GeneratePercept` pourra être exploité judicieusement.
    2. `gp` : soit `None` soit une instance de `GeneratePercept`. Aucun contrôle sur la valeur de gp autre que savoir s'il vaut `None` ou pas.
    3. `lCap` : la liste des capteurs de l'aspirateur. Par défaut lCap est initialisé à [ ].

- Un attribut en lecture/écriture `energie` qui sera toujours entre 0 et 100
- Un attribut en lecture/écriture `cpt` qui fonctionne entre 0 et le nombre de gènes (-1) du programme génétique. Ce compteur sera **utile** dans le cas des aspirateurs sans capteur.
- Un attribut en lecture/écriture `vivant` qui sera de type **bool** et de valeur **True** et passera à **False** quand `energie` est à 0
- Un attribut en lecture seule `program` contenant le ProgramGenetic reçu par le constructeur – ou créé par le constructeur
- Un attribut en lecture seule `nbTours` pour connaître le nombre d'itérations pendant lesquelles l'aspirateur était vivant.
- Une méthode `reset` qui, a minima remettra l'attribut `vivant` à **True** et l'attribut `cpt` à zéro.
- **getEvaluation**
- **getDecision**

2.2 Monde_AG

Il faudra redéfinir les méthodes et attributs

1. **getPerception**
2. **applyChoix**
3. **perfGlobale**