Fiche TP02-B

marc-michel.corsini@u-bordeaux.fr

21 mars 2016

En route pour la recherche des « programmes génétiques optimaux ».
On va considérer différentes situations toujours dans un monde $1 \times n$:

1. Deux sortes d'objets 0 (rien), 1 (poussière), un agent sans capteur,
2. Trois sortes d'objets 0 (rien), 1 (poussière), 2 (prise), un agent avec capteurs [6,8], [2,8] et [6,2,8]. Maximum 3 prises pour $n > 3$,
3. Plus d'objets, 3 jouets aspirables, 4 jouets déplaçables (si la case suivante est vide, sinon ne bouge pas) pour des mondes de taille $n > 5$, au plus 1 objet de type 3, au plus 1 objet de type 4. Pour tout type d'aspirateur (sans et avec capteurs).
4. Pour les aspirateurs avec capteurs, possibilité de panne d'un capteur sur les k (si un capteur est en panne à l'instant t les autres capteurs sont fonctionnels). Probabilité de panne 1 chance sur 10 (à chaque itération), la panne se traduit par une information à -1. **Remarque** le capteur 8 ne tombe jamais en panne.

Pour les aspirateurs sans capteur - la longueur de la chaîne est fixée à 20 gènes. Pour les aspirateurs avec capteurs, la longueur de la chaîne est déterminée par le nombre de cas possibles.
La fonction à optimiser est la même pour tout type d'aspirateur :

$$f = \frac{\text{energie}}{100} + \frac{\text{nettoyees}}{\text{sales}} - \frac{\text{nb Repos}}{\text{nb actions}} + \frac{\text{objets aspirables apres}}{\text{objets aspirables avant}} + \frac{\text{nb tour vivant}}{\text{nb tour max}}$$

Le simulateur, doit permettre de mettre tous les agents dans la même situation, pour garantir cette propriété on va stocker dans un fichier les environnement de simulations. Le score de fitness d'une chaîne sera la somme des f obtenues au cours de chaque simulation. Une ligne du fichier sera de la forme

```
type nbCol v1 .... vn p1 .. pk
```

type sera à valeur dans 0..2 : 0 que pour aspirateur sans capteur, 1 que pour aspirateur avec capteurs, 2 pour les deux types d'aspirateurs. v1..vn seront les objets dans la case 1 ... n de la table au début de la simulation. p1 .. pk seront k positions de départ pour l'agent dans ce monde les pj seront à valeur dans 0..nbCol-1.

1 Simulateur

Le constructeur reçoit en entrée le nombre d'itérations maximum pour une simulation, le fichier dans lequel il doit lire les configurations, les capteurs à utiliser [ ], [6,8], [2,8], [6,2,8], le fait que les capteurs peuvent tomber en panne ou pas (un booléen, `True` signifiant panne possible).
Par défaut, la liste des capteurs est initialisées à `[ ]`, le paramètre de panne à `False`
Le `constructeur` se contente de sauvegarder dans des attributs privés les paramètres reçus, aucun contrôle n'est effectué sur la validité des informations – une correction automatique est faite pour panne.
`Simulateur` dispose d'un attribut en lecture / écriture (`panne`) qui sera un booléen, et qui **doit** avoir la valeur `False` si pas de capteur.
`Simulateur` dispose d'une seule méthode `run` qui prend en entrée un programme génétique et un `GeneratePercept` :

```
Aucune vérification n'est faite sur les informations
Cree un aspirateur de type Aspirateur_PG avec les bons paramètres
Initialise les variables utiles (ajoute à l'aspirateur l'attribut panne)
Pour chaque monde
```

```
   Pour chaque position
       remplit la table, place l'agent
       fait la simulation
       ajoute le score f de la simulation
renvoie le score total
```

1.1 Aspirateur

Rien à faire la classe `Aspirateur_PG` répond aux besoins

1.2 Monde

Là il va falloir faire un petit peu de travail, la méthode de simulation ne satisfait pas tout à fait nos besoin. En effet on ne veut pas que l'initialisation soit aléatoire mais au contraire qu'elle respecte les données fournies par un tiers (la table et la position de l'agent).
On va donc définir la classe `MondeSimulation` qui dérive de la classe `Monde`.

1. Redéfinir la méthode **simulation** qui va recevoir 3 paramètres (dont deux avec valeurs par défaut `None`)
2. Redéfinir la méthode **initialisation** qui va recevoir 2 paramètres (avec valeur par défaut `None`)
3. Redéfinir la méthode **getPerception** pour traiter le cas d'un agent pouvant tomber en panne
4. Redéfinir la méthode **applyChoix** pour prendre en compte les nouveaux objets et les incidences des actions sur ces objets
5. Redéfinir **perfGlobale** pour être en accord avec les besoins de la simulation.

Le code va beaucoup ressembler à celui écrit pour le TP02-A

2 Algorithme Génétique

Le module est constitué de deux fichiers

1. `base_agslib` contient la gestion des chaînes. Pour manipuler une chaîne il faut spécifier la taille de celle-ci, l'alphabet qui est utilisé et la fonction d'évaluation.
   Pour créer un individu vous travaillerez à partir de la classe `Individu` – la classe `SomeOne` fournit les fonctionnalités mais ne doit pas être utilisée directement.
   De fait vous ne devriez pas accéder à ce fichier, sauf à titre consultatif.
2. `agslib` contient tout ce qui est nécessaire pour la manipulation d'une population. Il faudra compléter / écrire les méthodes manquantes en vous appuyant sur les commentaires. Les méthodes à réécrire sont :
   (a) **run** qui prend un nombre d'itérations, un fichier de sortie pour enregistrer le meilleur chromosome, le code de la méthode de sélection (0 roue de la fortune, 1 méthode des fractions)
   (b) **hasConverged** qui prend en entrée un numéro de gène et qui renvoie True si on a trouvé un gène convergent à cette position
   (c) **isOver** qui renvoie vrai si on a convergence de suffisamment de gènes pour que la population soit considérée comme convergente.
   (d) **_selectWheel** (méthode la roue de la fortune) renvoie une liste d'individus dont le nombre est la taille de la population
   (e) **_selectFraction** (méthode des fractions) renvoie une liste d'individus dont le nombre est la taille de la population.

3 Utilisation

Ne reste maintenant que l'étape permettant de faire le lien entre le simulateur et l'algorithme génétique. C'est le propos du fichier `main_simulator` dans lequel sont assemblés les différents outils secondaires pour ce TP.