Evolution artificielle et robotique

Fonction fitness, algorithme (1+1)-ES, robotique évolutionniste

Mise à jour : Mars 2017

Rendez-vous sur la page http://pages.isir.upmc.fr/~bredeche/ - onglet 3i025 pour télécharger le code source Python nécessaire pour faire les exercices ci-dessous (dépendances: Pygame, Matplotlib).

Recherche aléatoire

Pour cet exercice, on vous demande d'implémenter la fonction fitness donnée ci-dessous. Celle-ci a pour but de favoriser les comportements qui se déplacent rapidement et en ligne droite (plus la valeur de *fitness* est grande, plus le comportement est efficace).

$$fitness = \sum_{t=0}^{evalTime} (v_t * (1 - v_r) * (minSensorValue))$$

Avec la vitesse de translation v_t, la vitesse de rotation v_r, et la valeur du senseur renvoyant la distance minimum *minSensorValue* (ie. le senseur le plus "activé", c'est à dire ici celui qui renvoie la distance la plus faible (= les plus proche d'un obstacle)).

Nous allons tout d'abord considérer la recherche de valeurs de paramètres pour un robot de type Braitenberg. La vitesse de translation v_t et la vitesse de rotation v_r seront calculées comme suit:

$v_t = p_1 * s_1 + p_2 * s_2 + ... + p_{n-1} * s_m + p_n$

$v_r = p_{n+1} * s_1 + p_{n+2} * s_2 + ... + p_{n+n-1} * s_m + p_{n+n}$

Avec $p_1, p_2, ..., p_{n+n}$ les paramètres et $s_1, s_2, ..., s_m$ les *m* différents senseurs. Chaque paramètre prend une valeur parmi trois: 0 (=pas de lien), 1 (= lien excitateur), et -1 (= lien inhibiteur).

Pour cet exercice, vous partirez du code fourni dans *multirobots_evolution_template.py* disponible sur la page de l'UE, qui implémente une recherche aléatoire. Modifiez le code pour maintenir une archive du meilleur jeu de paramètres testés jusqu'ici, que vous mettrez à jour lorsque c'est nécessaire. Chaque jeu de paramètres doit être évalué pendant 200 itérations, et après 300 essais, vous devez afficher le meilleur comportement obtenu jusqu'ici, en boucle (i.e. vous réinitialiserez la position du robot toutes les 200 itérations, jusqu'à interruption du programme par l'utilisateur).

Remarques:

1. Les vitesses de translation et de rotation que vous donnez au robot ne sont pas forcément les véritables vitesses de rotation et translation (ie. la vitesse réelle de translation est 0 si le robot est bloqué). Pour savoir si un robot se déplace, vous pouvez par exemple utiliser l'évolution de la position au cours du temps.
2. vous allez devoir créer de nouvelle variable dans la classe Agent, en particulier, vous devrez stocker les paramètres du robot courant (pour utilisation immédiate) ainsi que les paramètres du meilleur robot testé jusqu'ici (pour archivage).
3. Le nombre d'itérations et le nombre d'essais sont des paramètres importants. De même que les conditions initiales. Pour obtenir un comportement robuste, il est préférable de varier les conditions initiales.

Algorithme (1+1)-ES

Le génome contiendra les paramètres *p* permettant de calculer les vitesses de translation et de rotation comme une combinaison des entrées, en ajoutant un paramètre de "biais" à chaque fois. C'est à dire:

$v_t = \tanh(p_1 * s_1 + p_2 * s_2 + ... + p_{n-1} * s_m + p_n)$

$v_r = \tanh(p_{n+1} * s_1 + p_{n+2} * s_2 + ... + p_{n+n-1} * s_n + p_{n+n})$

Avec $p_1, p_2, ..., p_{n+n}$ les paramètres à régler et $s_1, s_2, ..., s_m$ les *m* différents senseurs. On applique la fonction *tanh* (tangente hyperbolique) pour borner les sorties entre -1 et 1 (ce n'est pas obligatoire mais très utile).

Voici l'algorithme (1+1)-ES, qui permet d'optimiser en ligne un comportement pour un robot seul (dans cet exemple, on *minimise* la performance -- dans la fonction fitness défini auparavant, on *maximise*). Pour commencer, *sigma* sera fixé (à une petite valeur, p.ex. 10^{-2}) et ne changera pas au court de l'évolution. Dans un deuxième temps, vous pourrez mettre à jour *sigma*.

Initialize $x \in \mathbb{R}^d$ et $\sigma > 0$
  while not terminate
    $x' = x + \sigma N(0, I)$
    if $f(x') \leq f(x)$
      $x = x'$
      $\sigma = 2\sigma$
    else
      $\sigma = 2^{-1/4}\sigma$

Implémentez cet algorithme dans un robot. A chaque nouvelle évaluation, le robot doit être replacé à sa position de départ. Par rapport à l'exercice précédent, le génome contiendra maintenant des valeurs réelles.

Pour mesurer les progrès de votre évolution, vous devrez afficher pour chaque évaluation la performance obtenue. Comme il s'agit d'un algorithme stochastique, plusieurs runs seront nécessaires. Vous pouvez aussi varier les positions et orientation initiale pour favoriser la robustesse du comportement obtenu.

Remarques:

1. *random.gauss(0,1)* vous permet de générer des nombres aléatoires issues de la distribution N(0,1). Pour mémoire: N(mu,sigma) = mu+sigma*N(0,1)
2. *math.tanh(x)* permet d'obtenir la tangente hyperbolique de *x*
3. Vous devrez probablement borner les valeurs de vos paramètres afin qu'ils n'"explosent" pas. Par exemple, pour commencer:
    - valeurs du génome entre -10 et +10 (tirage au hasard à l'initialisation)
    - sigma entre 10^{-7} et 10^{-1} (valeurs exactes à définir)

Remarques par rapport au projet: *il est tout à fait possible d'entraîner, avec un (1+1)-ES (ou même avec une recherche aléatoire), des comportements dédiés à des situations particulières, puis de les organiser comme modules de comportements dans une architecture de contrôle (par exemple, subsomption ou arbres de comportements). Une architecture de contrôle peut ainsi très bien arbitrer entre des comportements écrits à la main et précédemment évolués.*