Algorithmes génétiques

I Présentation

Pour commencer, les algorithmes génétiques sont des algorithmes qui vont évoluer avec le temps, se modifier pour être plus performants en s'inspirant de l'évolution naturelle. Ces algorithmes vont simplement tester plusieurs possibilités pour n'en garder que les meilleures, évoluer et réessayer jusqu'à atteindre une solution convenable. Ils peuvent alors être très utiles dans des problèmes d'optimisation ou il est facile de mesurer la "qualité" d'une réponse alors qu'il est difficile de trouver la réponse idéale.

Dans un premier temps, notre algorithme va comporter 4 étapes qui seront répétées jusqu'à obtenir une solution qui nous convient :

- la création d'une nouvelle *population* à partir de la *génération* précédente
- la *mutation* de cette *population* afin de tester de nouvelles solutions
- l'attribution d'un score pour chacune des différentes solutions aportées
- la *sélection* des meilleurs *individus* de la population

Ici nous ne nous intéresserons qu'à un problème simple qui consiste à deviner une phrase (en créant des phrases aléatoires, le but est de trouver la solution "à taton"). Pour cela on définit une variable
`phrase = "The answer to the ultimate question of life, the universe and everything is 42."` (79 caractères), ainsi qu'une variable
`lettres = "azertyuiopqsdfghjklmwxcvbnAZERTYUIOPQSDFGHJKLMWXCVBN1234567890 ,;:.?!"` (69 caractères)(l'essentiel est que toutes les lettres de la phrase soient présentes). Le programme va donc piocher dans `lettres` des lettres pour former une phrase et retrouver `phrase`.

Dans la suite du programme, nos individus seront une liste composée de deux éléments : le score de l'individu ainsi que la phrase qui lui est associée (sous la forme d'une liste de caractère) par exemple : `individu = [10, ['r','e','p','o','n','s','e']`
Nous allons aussi définir une liste `parametres` former des constantes suivantes (dans cet ordre là) :

- tauxDeMutation (la probabilité de modifier une lettre)
- tauxDeMutationAdditive (la probabilité d'ajouter une lettre)
- tauxDeMutationSoustractive (la probabilité d'enlever une lettre)
- tauxDeSelection (la probabilité qu'un individu soit séléctionné)
- tailleDeLaPopulation
- generationMax (le nombre maximum d'essais possibles)

II L'algorithme

1. En suposant que l'on ait pas accès à la taille de la phrase à deviner, déterminer le nombre de comparaisons que devrait faire un algorithme naïf. Donner un ordre de grandeur du temps d'exécution, cela est-il raisonnable ?

2. Écrire une fonction `cree_population(parametre)` prenant la liste `parametre` en argument et renvoie une liste de longueur *tailleDeLaPopulation* contenant des individus generés aléatoirement (leur score sera initialisé à `None` et leur phrase ne comportera qu'un seul caractère (on poura utiliser la fonction `choice` du module `random` qui renvoie un élément aléatoire d'une liste passée en argument).
3. Créer une fonction `muttation_soustracive` qui prend en argument un `individu` et, si la phrase qu'il propose n'est pas vide, supprime le dernier caractère.
4. Créer une fonction `mutation_additive` qui prend en argument un `individu`, qui va ajouter l'un des caractères de `lettres` à la fin de la phrase proposée par `individu`.
5. Écrire une fonction `mutation_aléatoire` qui prend en argument un `individu` ainsi que la liste `parametre` et qui, pour chaque lettre de la phrase de l'individu va modifier la lettre avec une probabilité de *tauxDeMutation* (on pourra s'aider de la fonction `random` du module `random` qui renvoie un flottant aléatoire de $[0, 1[$.
6. Écrire alors une fonction `mutation`, prenant un `individu` et la liste `parametre`, qui modifie aléatoirement les lettres de l'individu, lui ajoute une lettre avec une probabilité de *tauxDeMutationAdditive* et lui en enlève une avec une probabilité de *tauxDeMutationSoustractive*.
7. Faire une fonction *reproduction* prenant en argument deux individus et crée un troisième individu avec un score initialisé à `None` et une phrase formée pour chaque caractère du caractère correspondant chez `individu1` avec une probabilité $p = \frac{s1}{s1+s2}$. (avec s1 le score de l'individu 1 et s2 celui de l'autre).(Si l'un des *parents* a une phrase plus longue que l'autre, *l'enfant* héritera de toutes les lettres excédantaires de ce dernier)
8. Faire alors une fonction `evaluation(individu)` qui va attribuer une note par effet de bord à un individu : pour chaque lettre correcte l'individu gagne 1 point. L'individu perd 10 points par lettre de trop ou lettre manquante. Les comparaisons sont bien entendu effectuées avec la variable `phrase`.
9. Créer alors une fonction `selection(population, parametres)` qui va trier la population en fonction des scores (on pourra utiliser le mot clef `sorted(population)` qui renvoie la liste triée), puis va séléctionner les individus avec une probabilité $p = \frac{k}{N} * \texttt{tauxDeSelection}$ (avec `k` le rang de l'individu dans la liste triée (k est petit pour les mauvais scores) et `N` la taille de la population). Et va renvoyer la liste ainsi créée (on veillera à ajouter le meilleur individu avec une probabilité de `1`).
10. Écrire alors la fonction `devine_la_phrase(parametres)` qui va générer une nouvelle population, la faire muter, l'évaluer et séléctionner les meilleurs individus pour créer une nouvelle population tant que la `phrase` n'a pas été trouvée, ou que l'on ait créé plus de générations que `generationMax`. Pour créer la nouvelle population, il suffit de choisir deux individus aléatoire parmis ceux séléctionnés, les reproduire, ajouter l'enfant dans la nouvelle population et recommencer jusqu'à ce que la nouvelle population soit pleinne.
11. tester votre fonction avec :
    — tauxDeMutation = 0.01
    — tauxDeMutationAdditive = 0.1
    — tauxDeMutationSoustractive = 0.1
    — tauxDeSelection = 0.4
    — tailleDeLaPopulation = 42

— generationMax = 20000

En comptant le nombre de générations créées, déterminer le nombre de comparaisons effectués. Comparer ce résultat à celui établi par l'algorithme naïf de la première question.