

Algorithmes génétiques

Un **algorithme génétique** est une méthode d'optimisation inspirée de l'évolution biologique (sélection naturelle, reproduction, mutation). Elle permet de rechercher une solution approchée à un problème, sans avoir à l'énumérer, en faisant évoluer une **population** de solutions candidates au fil de **générations** successives.

Dans ce TP, on va construire pas-à-pas un algorithme génétique dont le but est de retrouver une chaîne de caractères cible (la **solution**), en partant d'une population de chaînes aléatoires. Le fichier fourni importe déjà le module **random** sous le nom **rd**, et contient les fonctions déjà écrites ainsi que les squelettes des fonctions à compléter.

Principe général :

- (i) On génère une population initiale de chaînes de caractères aléatoires (les **chromosomes**), toutes de la même longueur que la solution cherchée.
- (ii) On évalue chaque chromosome par un **score** : le nombre de caractères correctement placés par rapport à la solution.
- (iii) On **sélectionne** les meilleurs chromosomes de la population.
- (iv) On les fait se reproduire par **croisement** (*crossover*), en recombinaison deux chromosomes parents pour obtenir deux enfants.
- (v) On applique une **mutation** aléatoire à certains enfants, avec une faible probabilité, pour introduire de la diversité.
- (vi) On recommence à l'étape 2 avec la nouvelle génération, jusqu'à obtenir un chromosome identique à la solution.

Rappels

Chaînes de caractères et indexation.

Une chaîne **s** se manipule comme une liste : **s[i]** renvoie le caractère d'indice **i**, et **len(s)** sa longueur. Le *slicing* **s[i:j]** renvoie la sous-chaîne allant de l'indice **i** (inclus) à l'indice **j** (exclu) ; **s[:j]** part du début, **s[i:]** va jusqu'à la fin. La **concaténation** de deux chaînes (ou de deux listes) se fait avec l'opérateur **+**.

```
"Bonjour"[2:5]      # -> "njo"
"Bon" + "jour"      # -> "Bonjour"
"Bonjour"[:3]       # -> "Bon"
"Bonjour"[3:]       # -> "jour"
```

Caractères et codes.

L'instruction **chr(n)** renvoie le caractère de code **n**, et **ord(c)** renvoie le code du caractère **c**. On travaillera avec des caractères de code compris entre 32 (" ") et 122 ("z").

Tirage aléatoire d'un échantillon.

On pourra utiliser l'instruction **rd.sample(L, k)**, qui renvoie une **nouvelle liste** de **k** éléments **distincts** tirés de **L sans remise** (la liste **L** n'est pas modifiée).

```
rd.sample([1, 2, 3, 4, 5], 2)  # -> par exemple [4, 1]
```

1 Génération de la population initiale

Question 1. Écrire une fonction **genere_chromosome** qui renvoie une chaîne de caractères aléatoire de longueur **len_chromosome** passée en argument, chaque caractère étant choisi aléatoirement parmi ceux de code compris entre 33 et 122 (bornes incluses).

Question 2. En utilisant la fonction précédente, écrire une fonction **genere_population0** qui prend en argument deux entiers **taille_pop** et **len_chromosome** et qui renvoie une **liste** de **taille_pop** chromosomes, chacun de longueur **len_chromosome**.

2 Évaluation d'un chromosome

Question 3. Écrire une fonction `score` qui prend en argument deux chaînes de caractères `chromosome` et `solution` (de même longueur) et qui renvoie le nombre d'indices où `chromosome` et `solution` coïncident.

Question 4. Que vaut `score(solution, solution)` ? Comment ce résultat permet-il de détecter qu'un chromosome est une solution du problème ?

3 Sélection des meilleurs individus

On souhaite trier la population par score décroissant, afin de pouvoir sélectionner les meilleurs chromosomes.

Question 5. Compléter la cellule **Q5** afin d'implémenter l'algorithme du tri rapide (*quicksort*) de la population selon le score de chaque chromosome de la population par rapport à `solution`.

Question 6. On souhaite maintenant sélectionner une partie de la population pour la reproduction, en conservant à la fois les meilleurs individus et un peu de diversité génétique. Compléter le code de la cellule **Q6** afin d'écrire une fonction `selection` qui prend en argument une liste de chromosomes `population` (de longueur N) et une chaîne de caractères `solution` et qui :

- (i) trie la population avec la fonction `tri_rapide`,
- (ii) conserve le premier tiers de la population triée (les $N // 3$ meilleurs chromosomes) dans une liste `best`,
- (iii) tire aléatoirement, **sans remise**, $N // 5$ chromosomes parmi le reste de la population dans une liste `rest` (on utilisera la fonction `rd.sample`),
- (iv) et renvoie la liste `best + rest`.

4 Reproduction : croisement et mutation

Question 7. Écrire une fonction `crossover` qui réalise un croisement entre deux chromosomes `ch1` et `ch2` en :

- (i) tirant aléatoirement un entier `i` compris entre 1 et $n - 1$ (bornes incluses),
- (ii) construisant deux chromosomes enfants en échangeant les portions situées avant l'indice `i` : le premier enfant est formé du début de `ch1` suivi de la fin de `ch2`, le second enfant du début de `ch2` suivi de la fin de `ch1`,
- (iii) et renvoyant le couple des deux enfants ainsi obtenus.

Question 8. Compléter le code de la cellule **Q8** afin d'écrire une fonction `mutation` qui prend en argument un chromosome `ch` et une probabilité `proba_mutation` (flottant) qui :

- (i) avec une probabilité `proba_mutation`, remplace **un** caractère de `ch`, choisi à une position aléatoire, par un caractère aléatoire de code compris entre 32 et 122,
- (ii) ou renvoie `ch` inchangé dans le cas contraire.

Question 9. Compléter le code de la cellule **Q9** afin de manière à définir une fonction `generation` qui construit une nouvelle génération de même taille que `population` selon le principe décrit au début du sujet (choix de parents, croisement, mutation éventuelle).

On ajoutera des enfants dans la nouvelle population jusqu'à ce que celle-ci ait atteint la même taille que la population de leurs parents.

5 Algorithme principal

Question 10. Écrire dans la cellule **Q10** une fonction qui prend en argument une liste **population** et une chaîne **solution** et qui renvoie le couple (**True**, **chromosome_solution**) si la liste **population** contient le chromosome **solution**, et le couple (**False**, **population[0]**) dans le cas contraire (le choix du chromosome renvoyé est arbitraire).

Question 11. À l'aide des fonctions précédentes, compléter le code de la cellule **Q11** de manière à écrire une fonction **resolution** qui prend en argument une **solution** et une probabilité **proba_mutation** (par défaut fixée égale à 0,2) et qui :

- (i) initialise une population de 100 chromosomes de longueur **len(solution)**,
- (ii) tant qu'aucun chromosome de la population n'est égal à **solution**, fait évoluer la population d'une génération à l'autre à l'aide de la fonction **generation**,
- (iii) compte le nombre de générations nécessaires,
- (iv) renvoie le couple (**chromosome_solution**, **nb_generations**).

Question 12. Tester la fonction précédente dans la cellule **Q12** sur **resolution("C'est Incroyable !")**. Combien de générations sont nécessaires ? Exécuter plusieurs fois cet appel ; le nombre de générations est-il stable ?

6 Pour aller plus loin

Question 13. Faites varier **proba_mutation** (par exemple 0.01, 0.2, 0.6) pour une même solution. Quel est l'effet d'une probabilité de mutation trop faible ? Trop élevée ?

Question 14. Dans la fonction **selection**, on conserve un tiers de la population sous forme des meilleurs individus, et on tire le reste aléatoirement parmi les autres. Pourquoi ne pas simplement conserver les meilleurs individus uniquement ?

Question 15. Proposer une modification de la fonction **score** qui pénaliserait davantage les grands écarts entre les caractères (par exemple en utilisant **abs(ord(car1) - ord(car2))**), plutôt que de compter uniquement les positions exactement correctes. Réécrire les fonctions **score** et **trouve_solution** en conséquence et discuter de l'effet attendu sur la vitesse de convergence de l'algorithme.