Ultra mastermind

Le Gourrierec Alan

December 20, 2023

Unité d'enseignement : Algorithmique et programmation impérative

Enseignant : Marteau P.-F.

Établissement : ENSIBS (Vannes)

Contents

Chapter 1

Spécifications

Pour la réalisation de ce projet, j'ai utilisé python et plus précisément la branche impérative de ce langage, car je suis plus familier avec cette dernière.

Je suis bien plus familier avec cette méthode de programmation. Le code n'est pas forcément le plus clair, c'est donc pour ca que ce dernier a été commenté et que les variables possède un nom, d'après moi, auto-porteur afin de comprendre leurs utilité.

Pour réaliser ce projet, j'ai utilisé la bibliothèque random de python et pour effectuer les diverses évaluations, j'ai utilisé les bibliothèques time, math et mathplotlib.

Chapter 2

Création des diverses fonctions

Pour réaliser ce projet, j'ai dû réaliser diverses fonctions et leurs tests unitaires, je vais donc vous présenter les plus importantes.

2.1 Proposition et Population

Ces deux fonctions sont très proches et servent plus ou moins à la même chose, c'est-à-dire créer une proposition pour résoudre le problème pour proposition (elle sert aussi à générer le code secret.).

```
def Proposition(size):
    return [random.randint(0, 255) for _ in range(size)]
```

Pour population, ceci est plus ou moins la même chose, mais il va créer une liste de taille "taille_pop" qui est une variable définie à l'initialisation du programme (libre à vous de la modifier).

```
def Population(taille_pop, size):
return [Proposition(size) for _ in range(taille_pop)]
```

2.2 Fitnesse et FitnessPop

Comme précédemment, ces deux fonctions sont plus ou moins les mêmes. La fonction fitness permet de déterminer, en comparant notre code secret avec la proposition, le nombre de lettres à la bonne position et le nombre de lettres à la mauvaise position.

```
def Fitness(mot_de_pass, proposition):
    GP = BP = 0  # GP est good_position et BP est bad_position
    for i in range(min(len(mot_de_pass), len(proposition))):
        if mot_de_pass[i] == proposition[i]:
            GP += 1
        elif proposition[i] in mot_de_pass:
            BP += 1
    return GP, BP
```

Quant à FitnessPop, sont objectif est de renvoyer une liste contenant la fitness de toutes les propositions de notre population dans le but d'après ne conserver que les meilleurs individus.

```
def FitnessPop(mot_de_pass, population):
return [Fitness(mot_de_pass, proposition) for proposition in population]
```

2.3 Eugenisme

Cette partie est une fonction nécessaire pour les algorithmes génétique. Elle permet de sélectionner les meilleurs éléments de notre population et de supprimer les plus mauvais. Nous appliquons pour ceci, un taux de conservation de cette population afin de conserver un certain pourcentage de cette dernière (il est de 60% dans mes tests).

```
1 def Eugenisme(population, evaluations, taux_conservation):
2     taille_conserver = int(len(population) * taux_conservation)
3     trie = [i for _, i in sorted(zip(evaluations, population), key=lambda couple
      : couple[0], reverse=True)]
4     return trie[:taille_conserver]
```

2.4 Reproduction et Mutation

Enfin, il ne nous reste plus qu'à effectuer les reproductions au sein de notre population. Pour ce faire, nous prenons une valeur aléatoire entre 0 et la taille de notre code secret pour savoir ou commence et ou fini la transmission des chromosomes du parentA au parentB. Nous obtenons ceci :

```
1 def Reproduction(parents, size):
2 cut_point = random.randint(0, size)
3 return parents[0][:cut_point] + parents[1][cut_point:]
```

Il faut ensuite générer des mutations, car si ceci n'est pas fait, nous resterons sur les mêmes solutions du début à la fin et ne pourrons pas trouver la solution. Pour ce faire, nous allons utiliser un taux de mutation, c'est-à-dire, une probabilité, pour chaque enfant, de voir un de ses chromosomes modifier.

```
1 def Mutation(enfant, taux_mutation):
2     for i in range(len(enfant)):
3         if random.random() < taux_mutation:
4             enfant[i] = random.randint(0, 255)
5     return enfant
```

Chapter 3

Coordination des nos fonctions

J'ai donc réalisé une dernière fonction nommé génétique, qui réutilise les précédentes dans le but de les faire fonctionner ensemble (c'est pour avoir un main quasiment vide). Cette dernière ressemble à ceci :

```
1  def Genetique(taille_pop, taux_mutation, taux_conservation, size):
2  t = time.time()
3  generation = 0
4  mot_de_pass = Proposition(size)
5  #print(f"Le code est : {mot_de_pass}\n")
6
7  pop = Population(taille_pop, size)
8
9  while True:
10 eval = FitnessPop(mot_de_pass, pop)
11 if mot_de_pass in pop:
12 print(f"Trouve au bout de la {generation + 1}eme generation")
13 print(pop[len(pop)-1])
14
15 parents = Eugenisme(pop, eval, taux_conservation)
16 enfants = [Mutation(Reproduction(parents, size), taux_mutation) for _ in range(
       taille_pop - len(parents))]
17
18 generation += 1
19 pop = parents + enfants
20
21 pop.sort(key=lambda x: Fitness(mot_de_pass, x)[0], reverse=True)
```

J'ai donc utilisé une boucle while qui est plus rapide qu'une boucle for (car l'algorithme est lourd, autant l'optimiser avec ceci).
La première partie est dédier à l'initialisation de notre population et de notre code secret (nommé "mot_de_pass").
La seconde partie, dédié à la résolution du problème dans un premier temps, évalue notre population avec la fonction FitnessPop (cf 2.2). Ensuite, nous allons modifier la liste des parents avec la sélection (cf 2.3), la reporduction et les mutations (cf 2.4).

Chapter 4

Analyse comportementale

Durant cette partie, nous allons observer les diverses manières pour tester les programmes que j'ai réalisés. Dans un premier temps, j'ai réalisé des tests en modifiant la taille du code secret, puis en modifiant le faut de conservation pour les populations et enfin le taux de mutation. Nous allons donc voir les courbes relatives au temps de ces dernières.

4.1 Variation de la taille

Dans un premier temps, nous allons faire varier la taille de 1 à 50 pour voir le temps que l'algorithme met à trouver la solution et aussi le temps que le nombre d'itérations que l'algorithme effectué pour chaque test. J'ai utilisé les paramètres suivants pour effectuer ceci :

- taille = x
- taux de conservation = 0.6
- taux de mutation = 0.05

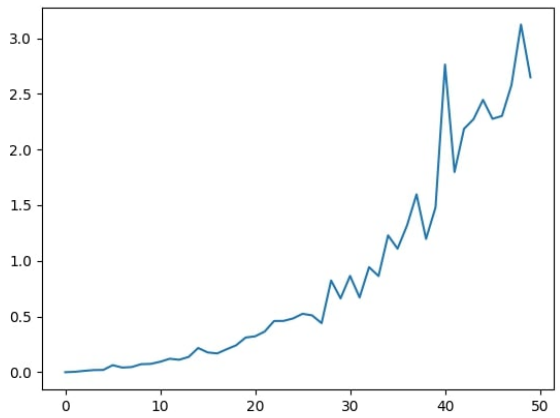

Figure 4.1: taille = f(temps)

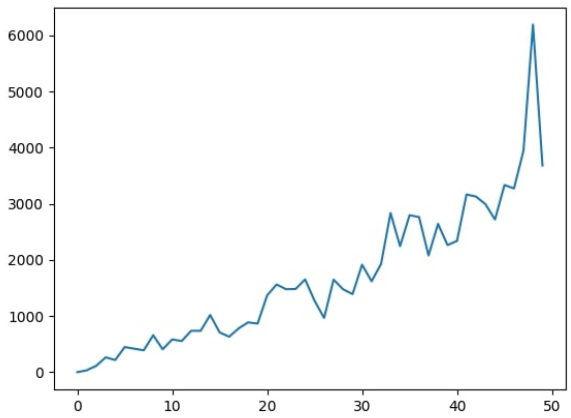

Figure 4.2: taille = f(iteration)

Il y a des imperfections sur les deux courbes, mais nous remarquons le caractère exponentiel de la courbe du temps et le caractère linéaire de la seconde.

4.2 Variation du taux de conservation

Nous allons maintenant faire varier le taux de conservation (je n'ai pas trouvé un meilleur nom, mais je ne voyais pas comment le nommer sinon). Pour ce faire, nous allons le faire varier de 0.1 à 0.9 (car le faire démarrer de 0 ou le faire finir à 1 n'as aucun intérêt) avec un incrément de 0.01 à chaque fois. Les autres paramètres seront :

- taille = 20
- taux de conservation = x
- taux de mutation = 0.05

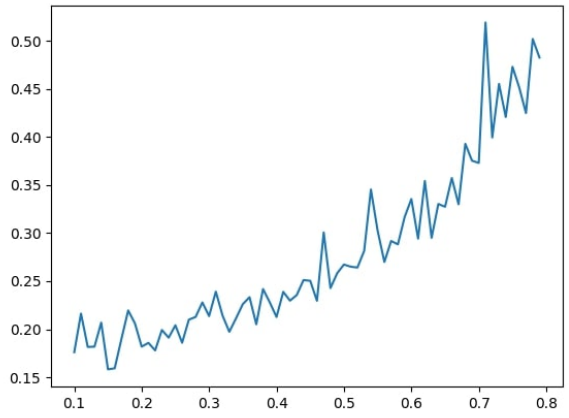

Figure 4.3: taux_concervation = f(temps)

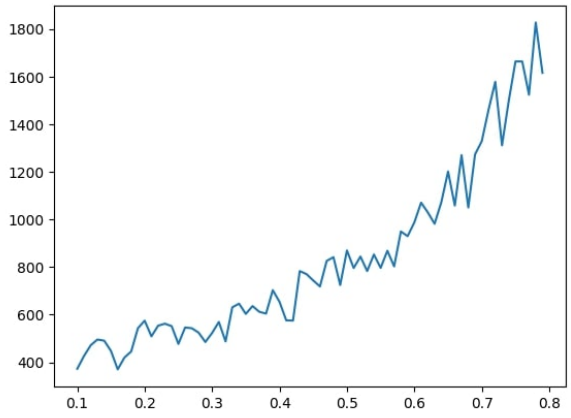

Figure 4.4: taux_concervation = f(iteration)

4.3 Variation du taux de mutation

Pour finir, nous allons faire varier le taux de mutation. Pour ce faire, nous allons le faire varier de 0.05 à 0.3 car comme nous allons le voir, ceci demande déjà beaucoup de temps pour les dernières itérations. Les paramètres seront donc :

- taille = 20
- taux de conservation = 0.6
- taux de mutation = x

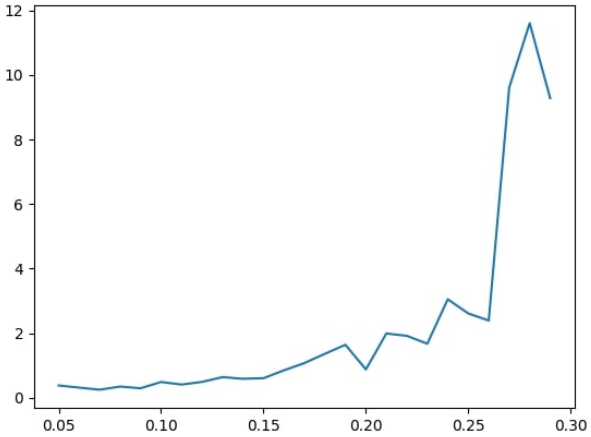

Figure 4.5: taux_mutation = f(temps)

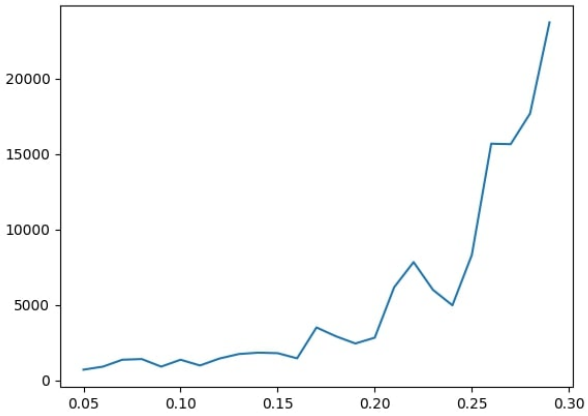

Figure 4.6: taux_mutation = f(iteration)

Nous voyons donc par ses graphes que le taux de mutation affecte beaucoup les performances de notre algorithme. Il faut donc le conserver suffisamment bas pour que notre complexité ne soit pas aberrante.

Conclusion

Ce projet a été très intéressant, je n'ai mal heureusement pas réussi à faire converger mon algorithme pour UMM++ ce qui me déçoit un peu. J'ai donc pu remarquer notamment au travers de la variation du taux de mutation, qu'un paramètre modifier légèrement, pourrais avoir des conséquences désastreuses sur la complexité d'un algorithme (cf 4.3).

Je pense, d'autre part que pour ceux n'ayant pas appris à programmer (notamment en python) au préalable auraient pu avoir beaucoup de mal avec le projet est peu guidé (ce qui est le but d'un projet).

Le projet est très pertinent, car nous avons pu découvrir le fonctionnement d'algorithme génétique.