Compte Rendu

Introduction

Le jeu de l'Ultra Master Mind consiste a deviner un phrase de taille donnée. Les seules informations a disposition sont si un caractère est bien placé, mal placé ou n'est pas dans la phrase.
Je vais dans ce compte rendu détailler une implementation pour résoudre le jeu.

Spécification et conception

Le jeu de l'Ultra Master Mind est un jeu qui peut être gagne par un algorithme genetique.
Un algorithme genetique est un algorithme qui cree une population d'individu, selectionne un echantillon selon un critere de selection, puis reproduit les individus, par rapport a 2 parents. Les nouveaux individus ont une chance de muter.
l'algorithme se repete sur un certain nombre de generations, au bout desquelles il trouve, ou non selon les cas, la phrase mystère.

notations

Nom	Description	Type
l	Taille de la phrase	int
mp	Phrase mystere	str
ng	Nombre max de generations	int
n	Taille de la population	int
ts	Taux de selection	float [0,1]
tm	Taux de mutation	float [0,1]

Spécification

Les individus sont appelés chromosomes dans ce sujet.
Un chromosome se compose de 2 parties distinctes :

- Un corps, qui contient la chaine de caracteres
- Une fitness, qui est evaluee au cours de l'algorithme, par rapport a la phrase mystere

procédure

```
generer ou rentrer la phrase mystere
genrer une population
```

```
tant que la phrase n'est pas trouvee et le nombre de generation max n'est
pas atteint :
        baisser le nombre de generation restantes
        evaluer la fitness des chromosomes
        si le 1er chromosome correspond a la phrase :
                la phrase est trouvee
                sortir de la boucle
        // reproduction
        trier les chromosomes
        selectionner les meilleurs chromosomes
        reproduire les chromosomes
si la phrase est trouvee :
        indiquer la phrase
sinon :
        dire qu'elle n'a pas ete trouvee
```

Organisation du code

Programme principal :

```
from ummLib import *
import matplotlib.pyplot as plt
print("#########################################################")
mode=input("type de prhase mystere (0:random, 1:rentree par l'utilisateur)
: ")

if mode=="0":
    l=int(input("entrez la taille de la phrase mystere : "))
    mp=generateMP(l)
elif mode=="1":
    mp=input("entrez la phrase mystere : ")
else:
    print("erreur de saisie")
    exit()

ng=int(input("entrez le nombre de generations max : "))
n=int(input("entrez la taille de la population : "))
ts=float(input("entrez le taux de selection (standard : 0.5): "))
tm=float(input("entrez le taux de mutation (standard : 0.1) : "))
type=int(input("entrez le type de fitness (1,2 ou 3 (selon le sujet du
projet)) : "))
print("#########################################################")

ngInit=ng
l=len(mp)
```

```
chroms=generateChromosomes(l,n)
print()

found=False
ngInit=ng
fitness_values = []
while not found and ng>=0:
    try :
        ng-=1
        for i in range (len(chroms)):
            fitness(chroms[i],mp,type)
        if chroms[0][1]==l or (chroms[0][1]==0 and chroms[-1][1]<0):
            found=True
            break

        # chroms=reproduct(chroms,mp,tm)
        print("generation {}".format(ngInit-ng))
        chroms= reproduct(chroms,ts,tm)
        fitness_values.append(chroms[0][1])
    except KeyboardInterrupt:
        print(ng)
        break

print("phrase mystere trouvee en {} generations ! \nla phrase mystere est
: {}".format(ngInit-ng,chroms[0][0])) if found else print("pas trouve
apres {} generations".format(ngInit-ng))
plt.plot(fitness_values)
plt.xlabel('Generation')
plt.ylabel('Fitness')
plt.show()
```

detail des fonctions de ummLib.py

```
def generateMP(l):
    """
    input : l, the size of the phrase
    output : a string of size l
    generate random mystery phrase of size l
    """
    phrase = ""
    for i in range(l):
        phrase += chr(random.randint(0, 255))
    return phrase
```

Usage : permet de generer aleatoirement une phrase de taille l.
Principe : ajoute l caracteres aleatoires les uns a la suite

```
def generateChromosomes(lc,n):
    """
    input : lc, the size of a chromosome
            n, the number of chromosomes
    output : a list of n chromosomes
    generate n random chromosomes of size lc
    index 0 is the chromosome itself, index 1 is the fitness
    """
    chromosomes = []
    for i in range(n):
        chromosome = ["",0]
        for i in range(lc):
            chromosome[0] += chr(random.randint(0, 255))
        chromosomes.append(chromosome)
    return chromosomes
```

Usage : permet de generer n chromosomes de taille l
Principe : ajoute a une liste de chromosomes, des chromosomes generes de la meme maniere que la phrase mystere. La fitness est initialisee a 0.

```
def fitness1(chrom, phrase):
    """
    input : chrom, a chromosome
            phrase, the mystery phrase
    output : the fitness of the chromosome
    Return the fitness of the chromosome chrom using -Σ(|C[i]-PM[i]|),
i=1..L"""
    fitness = 0
    for i in range(len(chrom[0])):
        fitness -= abs(ord(chrom[0][i]) - ord(phrase[i]))
    chrom[1] = fitness

def fitness2(chrom, phrase, alpha):
    """
    input : chrom, a chromosome
            phrase, the mystery phrase
    output : the fitness of the chromosome
    Return the fitness of the chromosome chrom using
#match+alpha.#Missed_placed"""
    fitness = 0
    match_count = 0
    misplaced_count = 0

    for i in range(len(chrom[0])):
        if chrom[0][i] == phrase[i]:
            match_count += 1
        elif chrom[0][i] in phrase:
            misplaced_count += 1
```

```
    fitness= match_count + alpha * misplaced_count
    chrom[1] = fitness

def fitness3(chrom, phrase):
    """
    input : chrom, a chromosome
            phrase, the mystery phrase
    output : the fitness of the chromosome
    Return the fitness of the chromosome chrom using  -
LevenshteinDistance(C,PM)"""
    chrom[1]=-Levenshtein.distance(chrom[0], phrase)

def fitness(chrom, phrase, type, alpha=0.5):
    match type:
        case 1:
            fitness1(chrom, phrase)
        case 2:
            fitness2(chrom, phrase, alpha)
        case 3:
            fitness3(chrom, phrase)
```

Usage : définir la fitness d'un chromosome. le type de fitness peut être sélectionné parmi celles listées dans le sujet.

```
def sortByFitness(chromosomes):
    """
    input : chromosomes, a list of chromosomes
    output : the list of chromosomes sorted by fitness
    sort chromosomes by fitness
    """

    chromosomes.sort(key=lambda x: x[1])

    chromosomes.sort(key=lambda x: x[1], reverse=True)
    return chromosomes
```

Usage : permet de trier une liste de chromosome par ordre de fitness décroissant

```
def select(chromosomes,ts):
    """
    input : chromosomes, a list of chromosomes
            ts, the selection rate
    output : the list of chromosomes selected
    select the best chromosomes to reproduce"""
    chromosomes = sortByFitness(chromosomes)
```

```
    n = math.floor(len(chromosomes)*ts)
    return chromosomes[:n]
```

Usage : selectionne les ts*n meilleurs chromosomes
Principe : trie les chromosomes, puis selectionne les ts*n premiers

```
def reproduct(chromosomes,ts,tm):
    """
    input : chromosomes, a list of chromosomes
            ts, the selection rate
            tm, the mutation rate
    output : the list of chromosomes reproduced
    select the best chromosomes to reproduct,
    reproduct the selected chromosomes, with a mutation rate of tm,
    can be parthenogenesis,
    also mutates some chromosomes
    """
    initPopSize = len(chromosomes)
    chromosomes = select(chromosomes,ts)
    print("best chromosome : {}".format(chromosomes[0]))
    newChromosomes = []

    while len(newChromosomes)+len(chromosomes) < initPopSize:
        choose1 = random.randint(0,len(chromosomes)-1)
        choose2 = random.randint(0,len(chromosomes)-1)
        parent1 = chromosomes[choose1]
        parent2 = chromosomes[choose2]
        cut = random.randint(len(parent1[0])//3,2*(len(parent1[0])//3))
        child = [parent1[0][:cut] + parent2[0][cut:],0]
        if random.random() < tm:
            mutation = random.randint(0, len(child[0])-1)
            child[0] = child[0][:mutation] + chr(random.randint(0, 255)) +
child[0][mutation+1:]
        newChromosomes.append(child)

    chromosomes+=newChromosomes
    return chromosomes
```

Usage : reproduire les chromosomes
Principe : Sélectionne les chromosomes.
Tant que le nombre de chromosomes et d'enfants n'est pas égale a la taille de la population initiale :
choisi 2 parents parmi les chromosomes, crée un enfant a partir de ces 2 parents, si un nombre aléatoire entre 0 et 1 est inférieur a tm, un caractère aléatoire est remplace par un autre.
ajoute les enfants aux chromosomes.

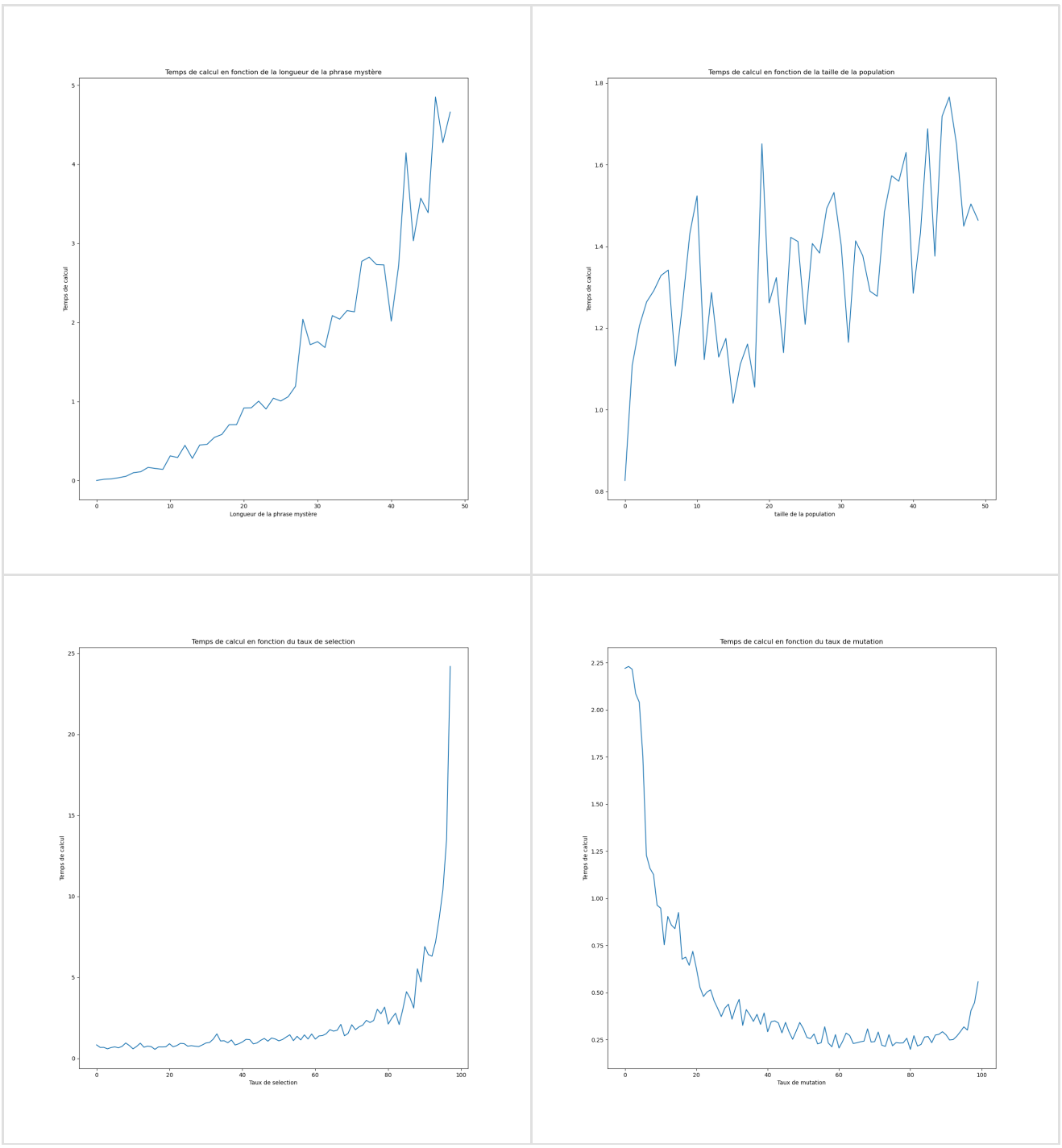
Analyse comportementale

Paramètres de base :

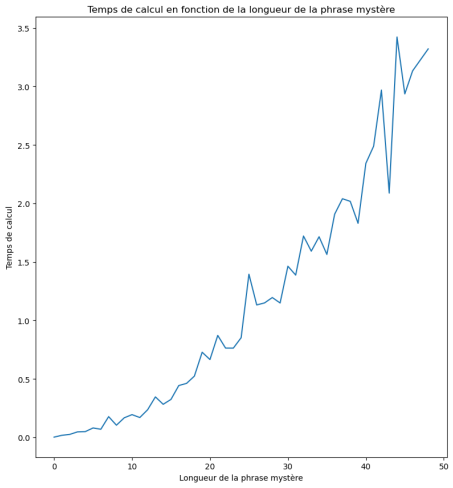
Nom	Description	Valeur
l	Taille de la phrase	25
n	Taille de la population	100
ts	Taux de selection	0.5
tm	Taux de mutation	0.1

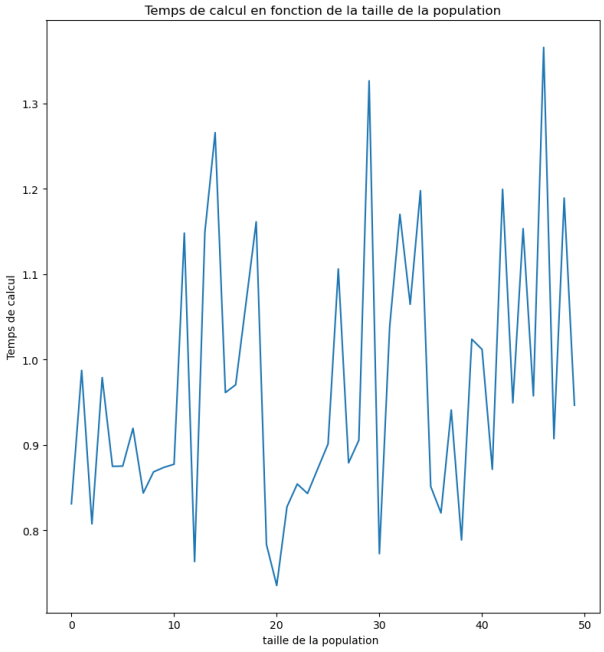
Pour éviter des temps d'execution infini, je limite le nombre de generations a 10^4

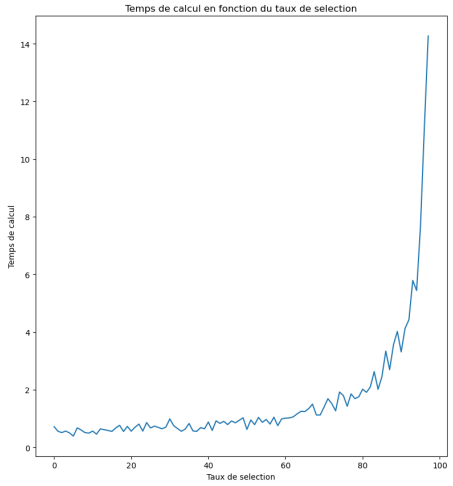
Fitness(C, PM) = -Σ(|C[i]-PM[i]|), i=1..L

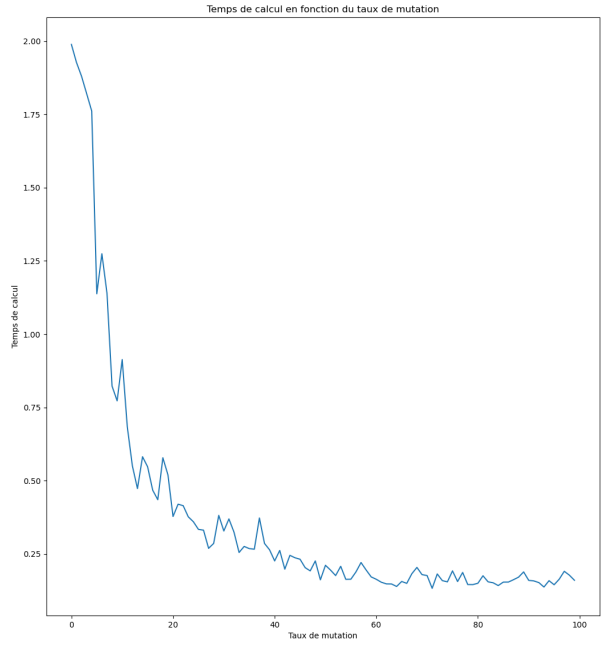


Fitness(C, PM) = #match+alpha.#Missed_placed

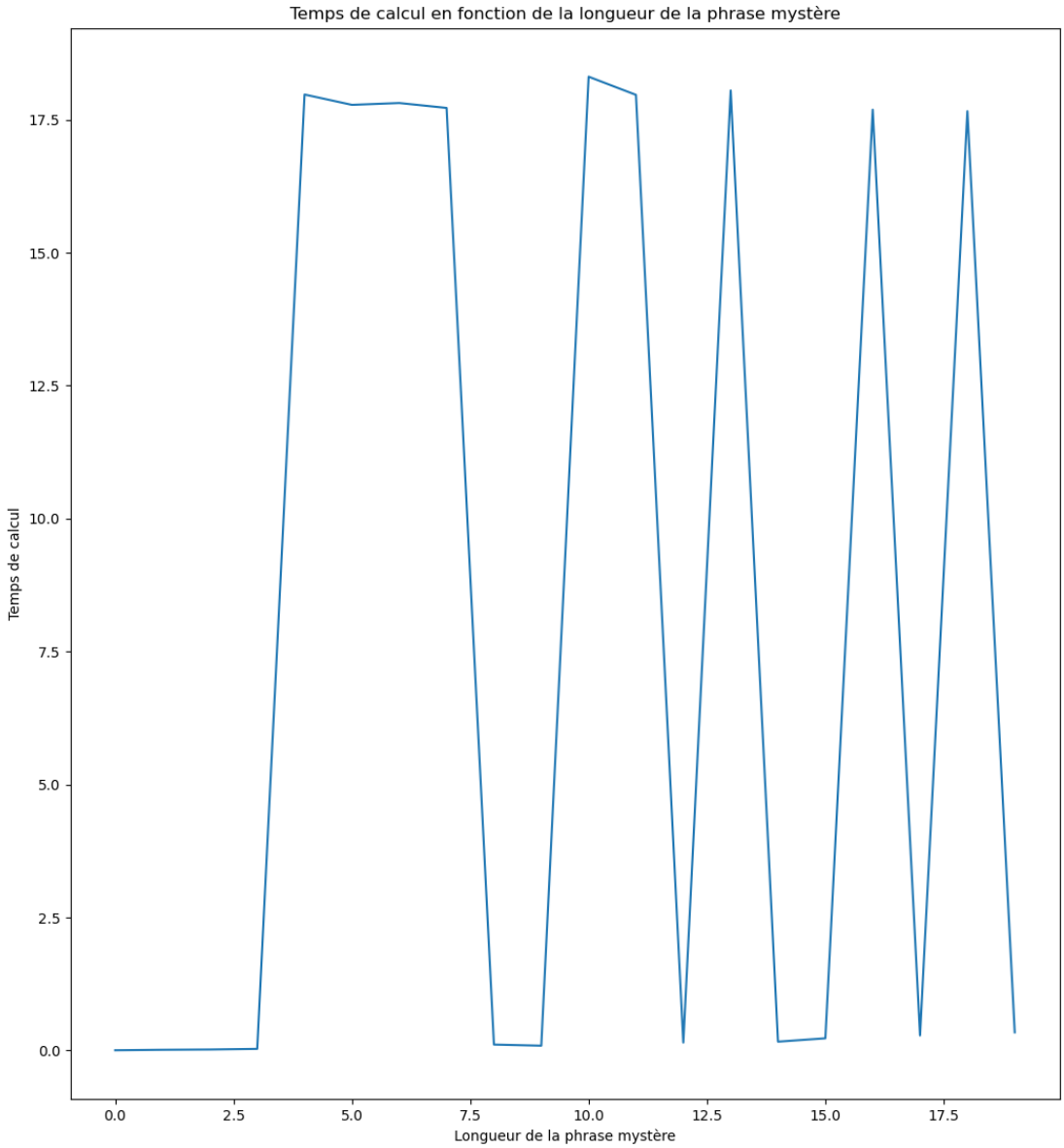








Fitness(C,PM) = - LevenshteinDistance(C,PM)



l'utilisation de la distance de Levenshtein entraine parfois un bloquage de l'algorithme, j'ai donc decide de stopper l'analyse a la 21e itteration sur la longueur, et de ne pas effectuer les autres tests avec cette methode.

Analyse

Le temps d'execution évolue de maniere quasi lineaire avec la taille de le phrase.
Les graphs pour la taille de la population sont extrêmement bruités. Nous ne pouvons pas conclure d'une influence sur le temps d'execution.
Le taux de selection a une influence exponentielle sur le temps d'execution.
Le temps d'execution semble evoluer de facon parabolique en fonction du taux de mutation, avec un minimum autour de 70%.

Installation

Dependances :

- matplolib
- Levenshtein

Utilisation :

Deposer ummLib.py et umm.py dans le meme dossier, dans ce dossier executer la commande : `python3 ./umm.py`
Remplir les informations demandees.
Enjoy.

Exemple d'execution

```
#########################################################
type de prhase mystere (0:random, 1:rentree par l'utilisateur) : 1
entrez la phrase mystere : Arch is the best OS (I use Arch BTW)
entrez le nombre de generations max : 1000
entrez la taille de la population : 100
entrez le taux de selection (standard : 0.5): 0.2
entrez le taux de mutation (standard : 0.1) : 0.7
entrez le type de fitness (1,2 ou 3 (selon le sujet du projet)) : 2
#########################################################

generation 1
best chromosome : ['`ën¿snA|ÙdÆË·Æ\x13\nf\x94\x9e8*}\x84ü\x04I×4±
¢)Îs¶\x9a\x91', 2.5]
generation 2
best chromosome : ['B\x00T\xa0\x87Þr¸\x17\x96¸IR\x95¡f--Í¾
èÖ[#°¥v¸SiVÑrÖb', 4.5]
generation 3
best chromosome : ['B\x00T\xa0\x87Þr¸\x17\x96¸IR\x95¤^
Xtd(\x1df\x18\x16\x0c\x9a(T\x06^\x88c¯ó3', 6.0]
generation 4
best chromosome : ['òÙãÓPù¥(\x13\x9c|O°p¤^ X\x87u(Üs[i°¥v¸SiVÑr\x01b',
6.5]
generation 5
...
...
generation 704
best chromosome : ['Arch is the best OS (I use Arch BTWb', 35.5]
generation 705
```

```
best chromosome : ['Arch is the best OS (I use Arch BTWb', 35.5]
generation 706
best chromosome : ['Arch is the best OS (I use Arch BTW)', 36.0]
phrase mystere trouvee en 707 generations !
la phrase mystere est : Arch is the best OS (I use Arch BTW)
```

Conclusion

Le taux de selection et la taille de la phrase font augmenter le temps d'execution.
le taux de mutation le fait diminuer, et la taille de la population ne semble pas avoir d'effet sur le temps d'execution.

Je pense que mon algorithme peut etre ameliore pour la version avec la distance de Levenshtein. En particulier sur la gestion de la fitness. En effet, l'algorithme a tendance a bloquer sur un certain chromosome, et tourner a l'infini.

Ayant change complement mon algorithme au cours du projet, je n'ai pas eu le temps d'implementer l'UMM++, ni le PGA.

Maxime Soulié