Projet
Ultra Master Mind (UMM)

Coutand Bastien - e2100676
3A - S1
2021 / 2022

ALGORITHME & PROGRAMMATION IMPÉRATIVE
3A - S1

Table des matières

1 Introduction

Dans le cadre de notre troisième année du cycle d'ingénieur en cybersécurité du logiciel au sein de l'ENSIBS. Il nous est proposé un projet de résoudre le problème de l'Ultra Master Mind (UMM), en exploitant un algorithme génétique [2]. Le problème est le suivant:

- On considère une chaîne de caractères de longueur L quelconque (phrase mystère).
- Les caractères sont des codes ascii codés sur un octet de 0 à 255 (alphabet de 256 caractères).
- Le jeu consiste à découvrir la phrase mystère. Le joueur soumet des phrases de longueur L, et le système répond en indiquant simplement le nombre de caractères en correspondance (match) et le nombre de caractères mal placés (miss placed).

2 Installation et utilisation

2.1 Procédure d'installation

- Etape 1: Décompresser le .zip ***"UMM_project_Coutand_Bastien"***
- Etape 2: Se placer dans le dossier ***"umm_project/"***

2.2 Lancement projet et tests

Projet + tests:

```
./run.sh
```

Les tests vont se lancer dans un premier temps, puis ensuite, un environnement virtuel sera créé pour exécuter le projet. Il sera supprimé à la fin.

3 Spécifications

3.1 Définition des termes

Théorie de l'évolution	**Algorithmes génétiques**	**Problème de l'umm**
Individu/chromosome	Une solution possible au problème	Une séquence
Population	L'ensemble des solutions étudiées	Les séquences à évaluer
Reproduction	Croisement de deux solutions pour en produire une nouvelle	Nouvelle séquence obtenue par combinaison de deux autres
Mutation	Modification aléatoire d'une solution	Permutation de certains gènes dans une séquence
Sélection	Élimination des solutions les moins adaptées	Élimination des séquences les moins proches
Gène	Élément d'une solution	Un élément de la séquence

Les constantes :

- L longueur des chromosomes (et de la phrase mystère) **[10 <= L]**
- N la taille de la population (nombre d'individus) **[10 <= N]**
- TS le taux de sélection (ou de reproduction) **[0.1 < TS <= 1]**
- TM le taux de mutation **[0.1 < TM <= 1]**
- NG le nombre de génération **[100 < NG < +inf]**

3.2 Versions utilisées

Python 3.9

3.3 Spécification des fonctions auxiliaires

Type	Méthode et description
string	`def str_input(msg:int, min:int, max:int)` Demande à l'utilisateur une chaîne de caractères selon la demande **msg**. Et vérifie que sa longueur est dans la plage **min** et **max**.
float	`def number_input(msg:str, mini:float, maxi:float, t)` Demande à l'utilisateur un entier selon la demande **msg**. Et vérifie que sa longueur est dans la plage **min** et **max**. **t** correspond au type voulu (int/float).
list	`def fusion_sort_decreasing(tab:list)` Trie de façon décroissante un tableau passé en entier avec la méthode de tri fusion.
list	`def fusion_sort_growing(tab:list)` Trie de façon croissante un tableau passé en entier avec la méthode de tri fusion.
list	`def extract_sub(tab:list, number:int)` Créer une liste des n° **number** des tuples de la liste **tab**.
string	`def list_to_char(list_to_trad:list)` Convertis la liste d'entiers comprise entre 0 et 255 (**list_to_trad**) en une chaîne de caractères. Utilisation de la Table ASCII étendue.

3.4 Spécification des fonctions génétiques

Type	Méthode et description
list	`def genese()` Créer une population initiale de taille **N**. Chaque individu sera de taille **L**. ou **L** = **PM**.
int	`def fitness_distance(C:list)` Fonction qui calcule la distance entre le chromosome **C** et la phrase mystère **PM**, avec la formule: *-Σ(|C[i]-PM[i]|), i=0..(L-1)*
int	`def fitness_max_weight(C:list)` Fonction qui calcule la somme des poids du chromosome **C** par rapport à la phrase mystère **PM**, avec la formule qui s'applique à chaque caractère: *0 <= #MissedPlaced(val:1) + #Match(val:9) <= 10*
int	`def dist_levenshtein(C:list)` Nous appelons distance de Levenshtein entre deux chaînes de caractères M et P le coût minimal pour transformer M en P en effectuant uniquement des opérations élémentaires (au niveau d'un caractère). Seulement des opérations élémentaires (au niveau d'un caractère). Opérations élémentaires : substitution / insertion (ou addition) / suppression (ou effacement).
list	`def reproduction(pop:list)` Deux individus P et M sont tirés au hasard parmi les individus sélectionnés. On tire au hasard deux points de coupures cut1 = **L** * (⅓) et cut2 = **L** * (⅔). On obtient un nouveau chromosome (individus) CM comme suit. CM = concatenation(P[:(1/3)], M[(1/3):(2/3)] + P[(2/3):]) Le nouvel individu est ajouté à la population : nous itérons la procédure afin de retrouver une population de **N** individus.

Type	Méthode et description
list	`def selection(pop:list)` Tous les individus de la population possèdent un chromosome. Les individus sont classés en fonction de la valeur fournie par la fonction Fitness. Les **TS** x **N** meilleurs individus sont sélectionnés pour la reproduction. Les autres sont supprimés, ils seront remplacés par les descendants créés par la reproduction. Elle permet le choix de la fonction fitness à utiliser, en fonction de la constante **FITNESS_CHOICE** (fichier ./utile/const/constante.py) Lim(C -> solution) `fitness_distance` = 0 (Trie décroissant) Lim(C -> solution) `fitness_max_weight` = +inf (Trie décroissant) Lim(C -> solution) `fitness_levenshtein` = 0 (Trie croissant) Lors de la sélection, on prend seulement les **TS** x **N** meilleurs individus, donc pour une meilleure efficacité, il faut avoir les meilleurs individus au début du tableau. D'où les différences de trie. Sinon cela oblige à parcourir tout le tableau.
None	`def mutation(pop:list)` On choisit au hasard **TM** x **N** individus sur lesquels sera portée la mutation d'un gène. Pour chaque individu sélectionné, on tire au hasard un gène (un caractère) parmi **L** et on modifie aléatoirement entre 0 et 255, sa valeur.
list	`def evolution(pop:list)` Sur plusieurs générations, jusqu'à trouver **PM**, on itère: - Évaluation des individus de la population - Sélection d'une partie de la population - Reproduction par croisement de certains individus - Mutation de certains individus

3.5 Spécification des tests

Les tests ont été effectués sur les fonctions génétiques, de tries et de transformations du fichier. Les tests sont visibles dans le dossier ***./test/***.

Pour les lancers, référez-vous au chapitre **2.2 - Lancement projet et tests**.

3.6 Spécification du programme de démarrage

Le programme de démarrage est divisé en 2 fichiers: le fichier ***main.py*** et le fichier ***run.py***.

Le fichier ***main.py*** permet le lancement d'une des deux fonctions du fichier ***run.py***. Soit la fonction ***run()***, qui exécute le programme principal. Soit la fonction ***start_run()*** qui demande à l'utilisateur s'il veut choisir ses paramètres ou utiliser ceux de bases notées dans le fichier ***utils.constantes.py.*** Elle lui demande aussi s' il veut afficher une courbe.

Type	Méthode et description
None	`def __start_run__()` Choix de départ de l'utilisateur: • Basic parameters: lance le programme avec les valeurs par défaut dans le fichier ***./utils/constante.py***. • Choose your parameters: Demande à l'utilisateur de choisir ses propres valeurs des paramètres du jeu (**PM**, **N**, **TS**, etc...). Affichage les paramètres du jeu et demande à l'utilisateur s' il veut tracer la courbe du nombre de génération sur les résultats de la fonction fitness choisi.
None	`def run()` Lance la boucle du jeu, calcule le temps d'exécution, affiche la courbe et demande à l'utilisateur s'il veut rejouer.

4 Organisation du code

4.1 Organisation globale

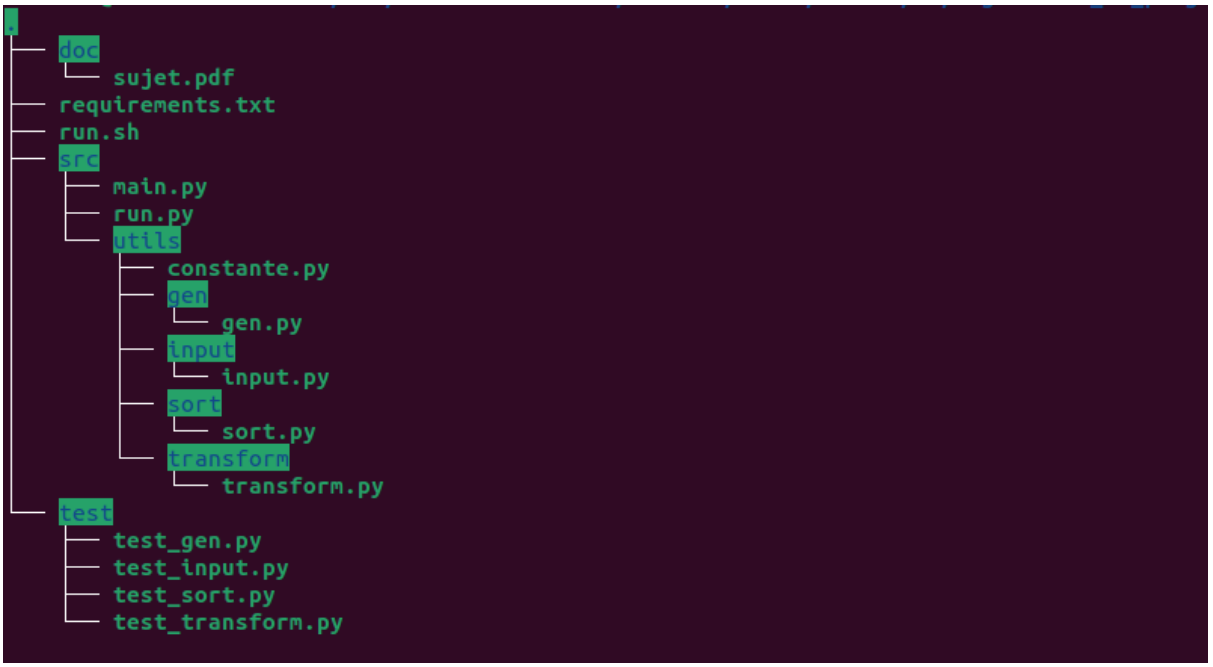

figure 1 - Arborescence du projet [4]

Le dossier ***doc***, contient le sujet du projet.

Le dossier ***src***, contient:

- Les sources du projet (les fonctions génétiques. Fichier ***gen.py***).
- Les fichiers de lancement du projet (fichiers ***run.py*** et ***main.py***).
- Les fichiers dit “auxiliaires” comme les fonctions de tries, d’entrées, etc... (dossier ***utile***).
- Le fichier avec les constantes du projet (fichier ***constante.py***).

Le dossier ***test***, contient:

- Les fichiers de tests (fichier ***test_ *.py***).

Le fichier ***run.sh*** qui permet de lancer les tests et le programme.

Le fichier ***requirements.txt*** qui comprend les librairies auxiliaires à installer (installer dans la ***run.sh***).

4.2 Diagramme des flux

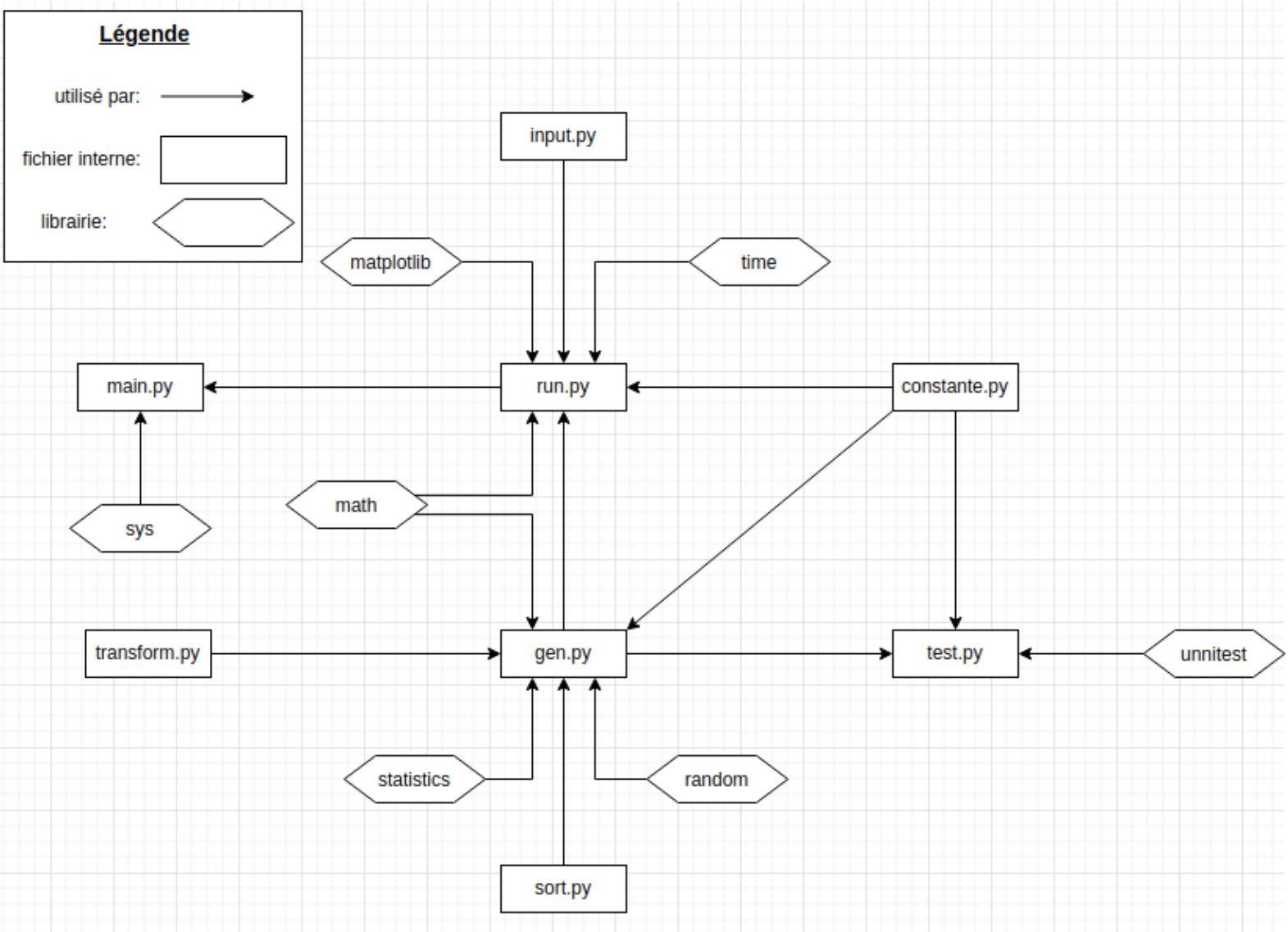

figure 2 - Diagramme des flux [5]

5 Analyses comportementales

5.1 Courbes

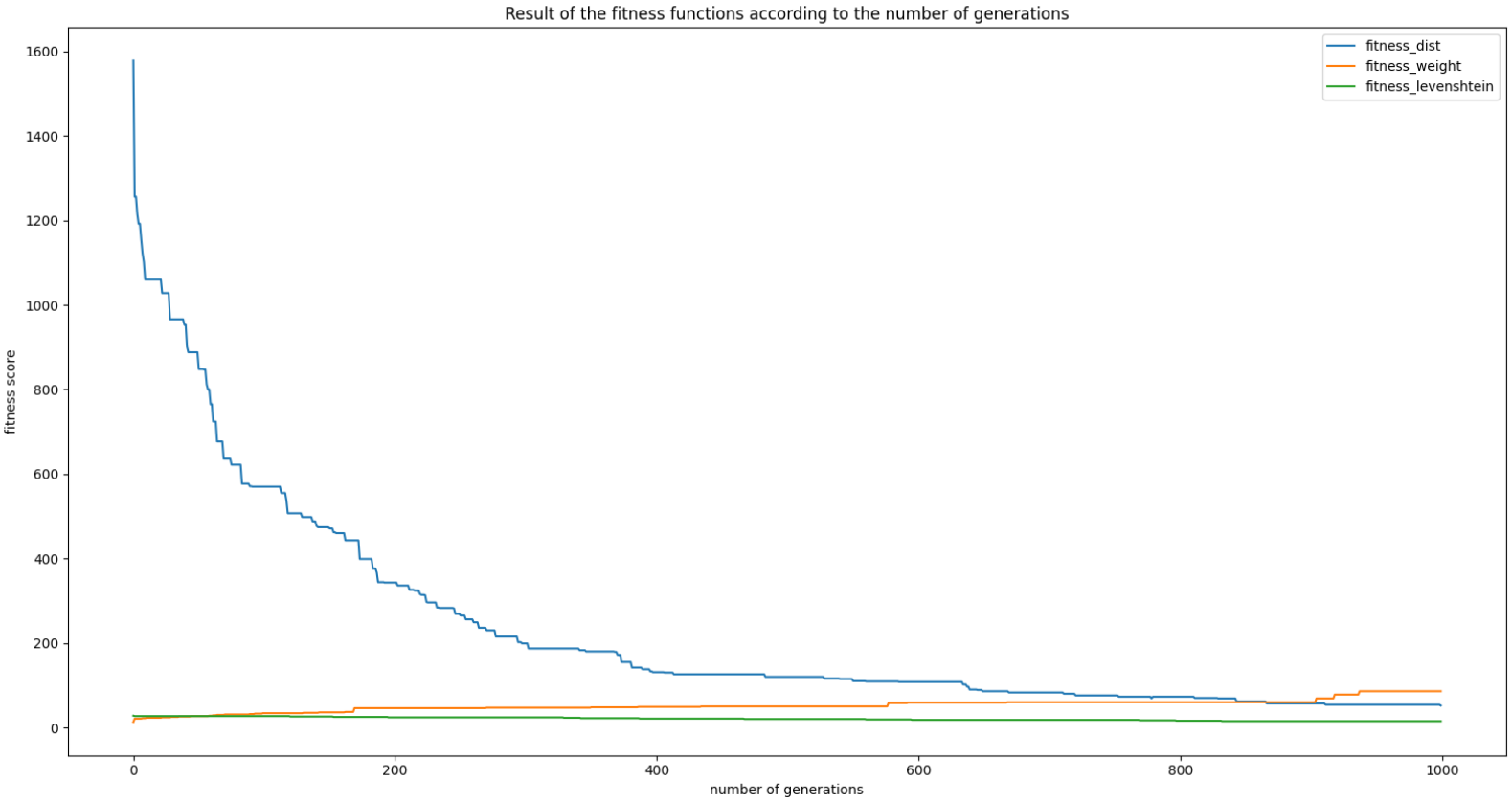

figure 3 - Graphique des scores des fitness [6]

```
time fitness_dist: 0.9424285888671875 s
time fitness_weight: 1.2787411212921143 s
time fitness_levenshtein: 26.391132354736328 s
```

figure 4 - Temps des fitness [7]

Valeurs par défaut: "PM": "Hello*I a^m a (tes5t phr$ase+", "L": 29, "N": 100, "TS": 0.3, TM": 0.05, "NG": 1000, "FITNESS_CHOICE": 1 (fitness_dist), "CURVE_CHOICE": 0

On observe via les temps que la fonction de distance est celle qui met le moins de temps à trouver la solution, suivi de peu par la fonction de poids et en dernier la fonction de distance de Levenshtein. La fonction de distance est donc la plus efficace des trois avec.

5.2 Variation des paramètres

Les graphiques suivants montrent l'évolution des temps d'exécution de l'UMM en fonction de la variation de certains paramètres. Chaque "barres" correspond à une moyenne de 10 exécutions avec la valeur du paramètre.

Valeurs par défaut: "PM": "Hello*I a^m a (tes5t phr$ase+", "L": 29, "N": 100, "TS": 0.3, TM": 0.05, "NG": 1000, "FITNESS_CHOICE": 1 (fitness_dist), "CURVE_CHOICE": 0

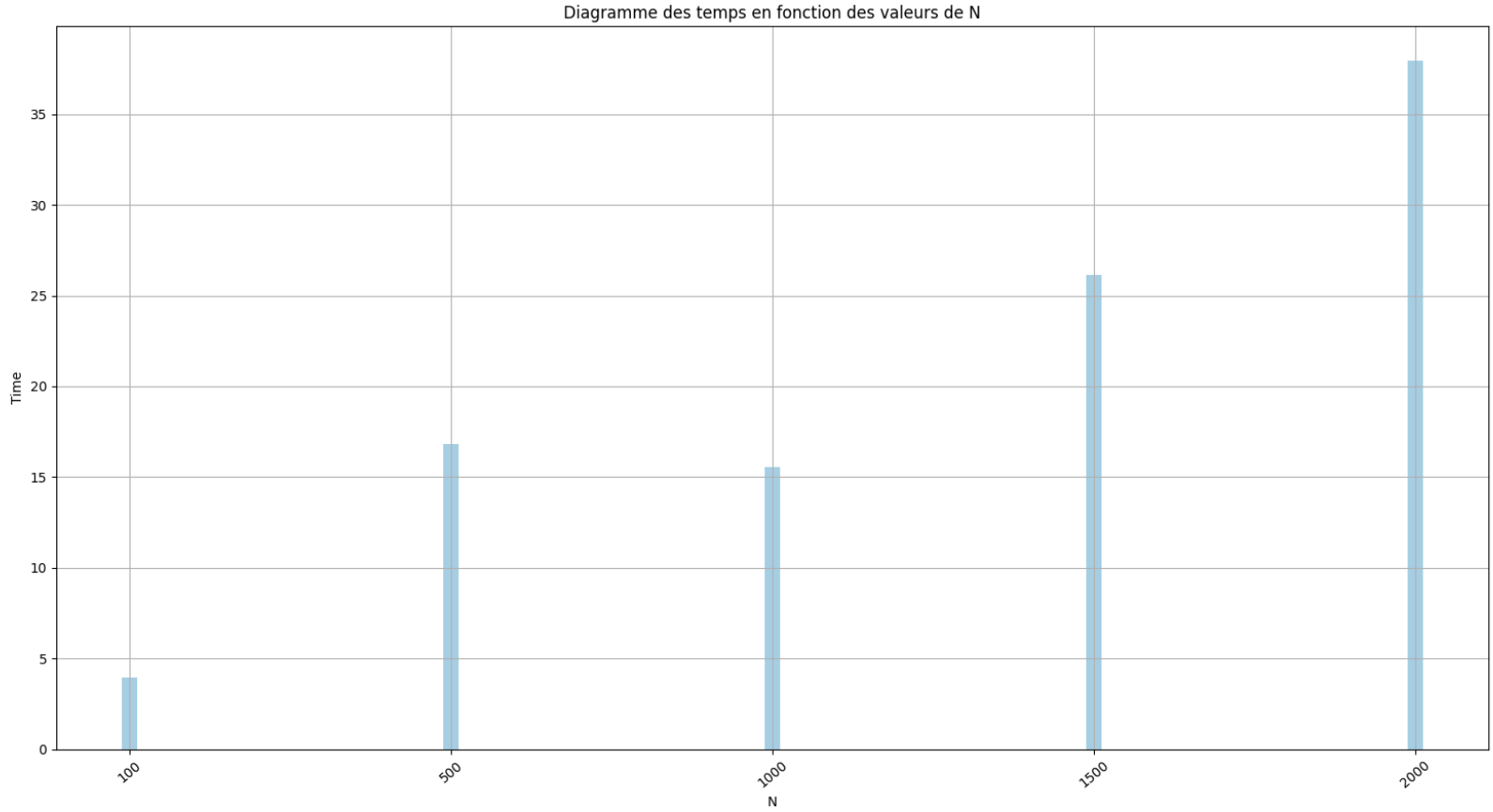

figure 5 - Temps de l'UMM en fonction du paramètre N [8]

En faisant varier le paramètre **N**. On observe que si **N** diminue alors il ya moins de générations. Ceci s'explique par le fait que si le taux de sélection est faible, alors on choisit les meilleurs de chaque population. Donc la reproduction donnera de meilleurs enfants. Et inversement.

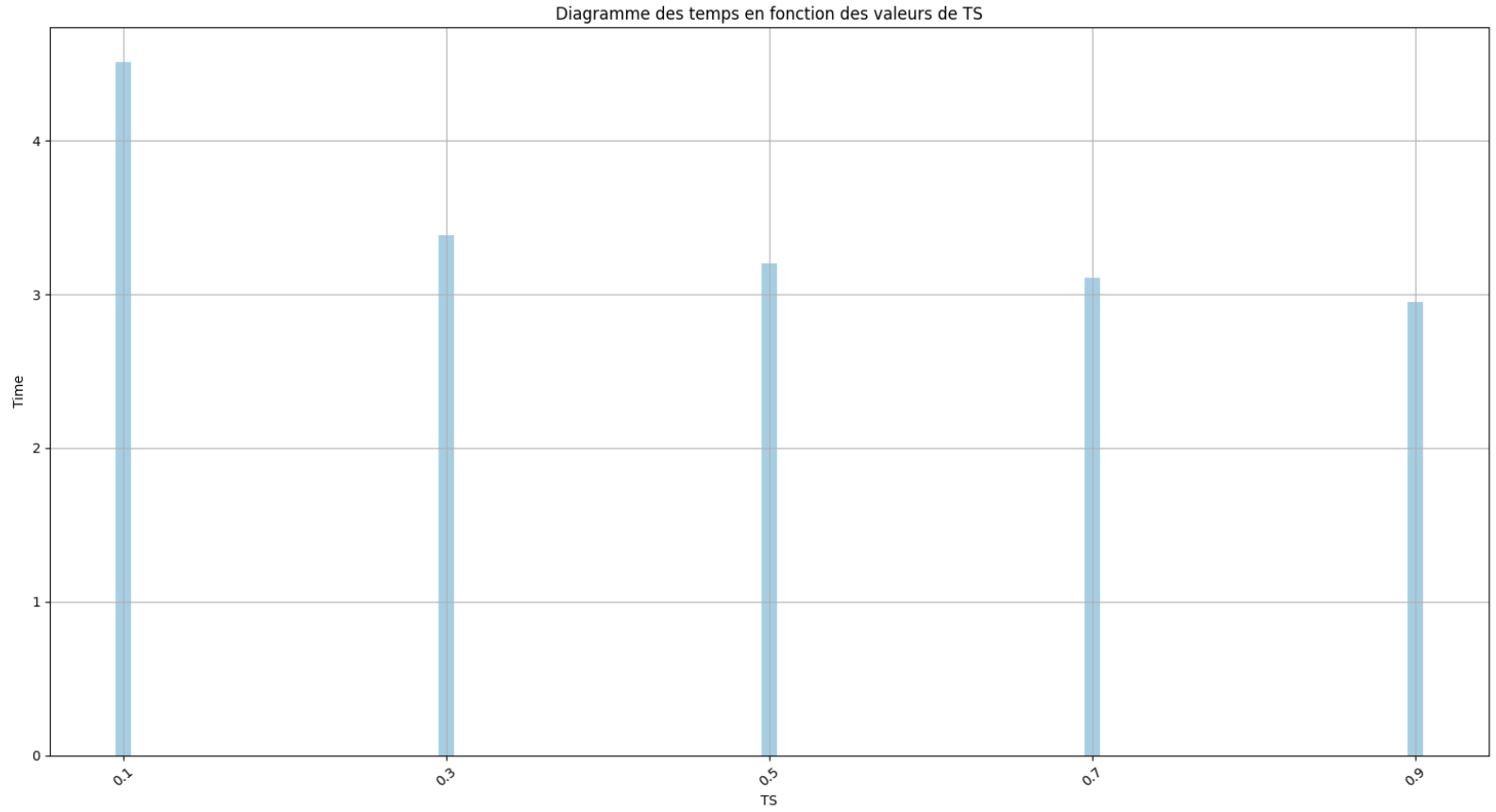

figure 6 - Temps de l'UMM en fonction du paramètre TS [9]

En faisant varier le paramètre **TS**. On observe que si **TS** diminue alors le temps et le nombre de générations diminuent. Ceci s'explique par le fait que si le taux de sélection est faible, alors on choisit les meilleurs de chaque population. Donc la reproduction donnera de meilleurs enfants. Et inversement.

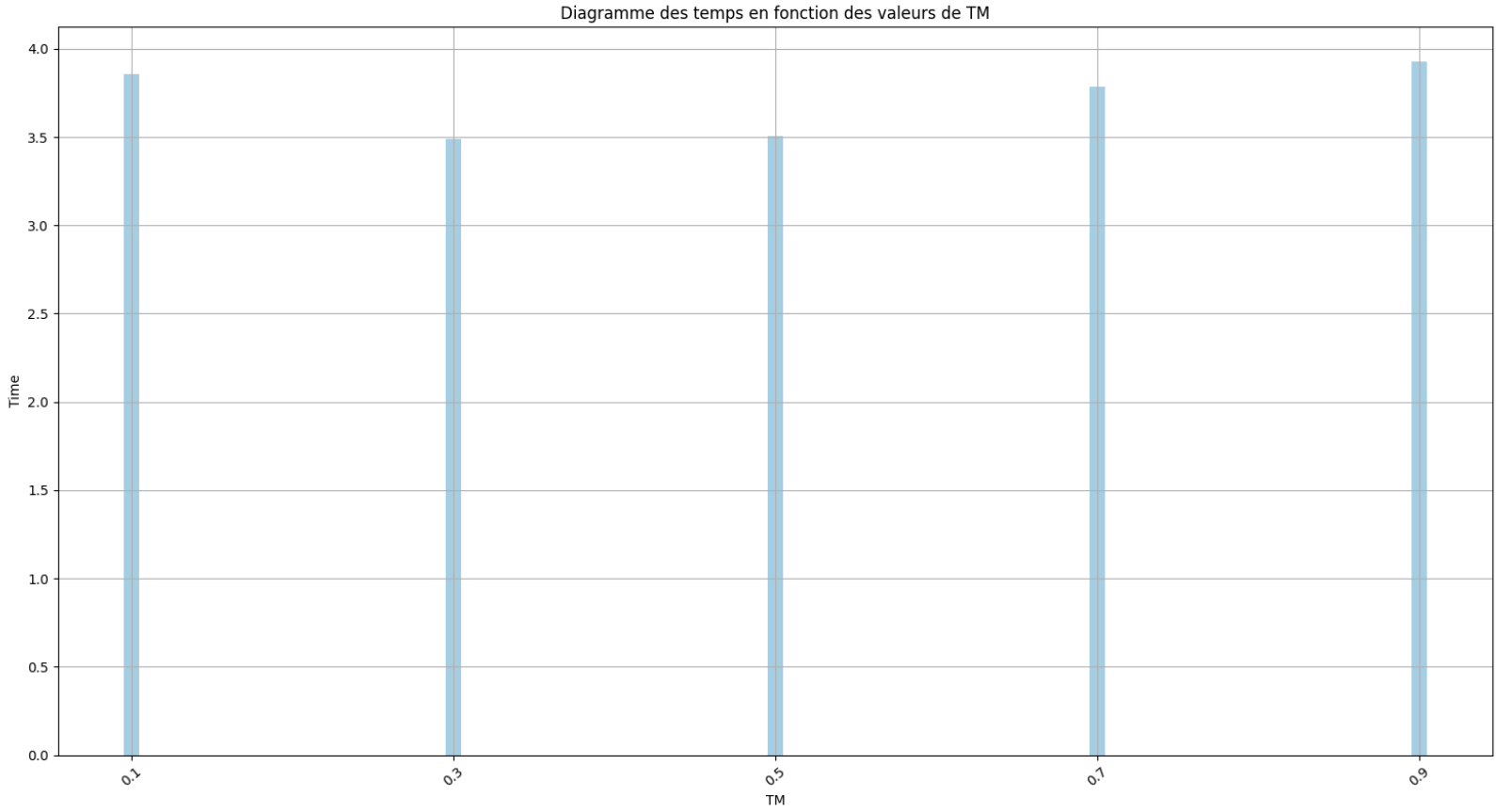

figure 7 - Temps de l'UMM en fonction du paramètre TM [10]

En faisant varier le paramètre **TM**. On observe que si **TM** augmente, alors le temps et le nombre de générations diminuent jusqu'au stade de **TM** = 0.5. Ceci s'explique par le fait que si le taux de mutation est grand, plus il y aura de nouveaux individus induits par cette mutation. Mais si le taux de mutation est élevé alors on ajoute plus de "bruit" que d'enfant correct possible. Et inversement.

5.3 Convergence

- **fitness_dist:**
  - domain: [-inf, 0]
  - $\lim_{C \to Solution}$ fitness_dist = 0
- **fitness_weight:**
  - domain: [0, +inf]
  - $\lim_{C \to Solution}$ fitness_weight = + inf
- **fitness_levenshtein:**
  - domain: [0, +inf]
  - $\lim_{C \to Solution}$ fitness_levenshtein = 0

5.4 Complexité empirique

- **fitness_dist:**

  Parcours de la liste de taille L, représentant le chromosome en question.

  Complexité O(L)

- **fitness_weight:**

  Pareil que fitness_dist.

  Complexité O(n)

- **fitness_levenshtein:**

  Parcours de la matrice de taille n et celle de taille m.

  Complexité O(n * m).

6 Conclusion

Le projet a été très intéressant. Mélanger la génétique et l'informatique permet de résoudre des problèmes dits complexes, avec des temps faibles.

En amélioration, on pourrait réaliser:

- Une fonction fitness de Levenshtein récursive fonctionnelle et rapide.
- Faire un programme plus "user-friendly".
- Mettre le programme sous la forme de thread pour pouvoir afficher les courbes plus rapidement.

De même, je pense que le sujet pourrait être élargi et étendu sur des solutions de brute forcing par exemple.

Le projet à été réalisé intégralement grâce à l'IDE pycharm et à l'outil de versionnage GIT (via GITHUB) et GitKraken, par Bastien Coutand.

7 Références

[1] Wikipédia. logo de l'ENSIBS - logo.
https://fr.wikipedia.org/wiki/%C3%89cole_nationale_sup%C3%A9rieure_d%27ing%C3%A9nieurs_de_Bretagne_Sud

[2] Algorithme génétique explications.
http://www-igm.univ-mlv.fr/~dr/XPOSE2013/tleroux_genetic_algorithm/fonctionnement.html

[3] Distance de Levenshtein de Wagner et Fisher.
https://128mots.com/index.php/2021/01/19/levenshtein-python/?amp

[4] Coutand. Arborescence du projet - image.

[5] Coutand. Diagramme des flux - image.

[6] Coutand. Graphique des scores des fitness- image.

[7] Coutand. Temps des fitness - image.

[8] Coutand. Graphique du temps d'exécution de l'UMM avec le paramètre N qui varie - image.

[9] Coutand. Graphique du temps d'exécution de l'UMM avec le paramètre TS qui varie - image.

[10] Coutand. Graphique du temps d'exécution de l'UMM avec le paramètre TM qui varie - image.