# Informatique Bio Inspirée – Algorithmes Génétiques

L’objectif de ce TP est de trouver un mot de passe en se reposant sur un score donné à chaque tentative.

Sujet : https://perso.liris.cnrs.fr/sforest/ibi/sujet.pdf
Code source : https://github.com/BruJu/IBI-Genetics

*Le code source que nous proposons a été écrit pour résoudre la blackbox de python3.7*

## Choix d’implémentation

### Génotype

Nous avons choisis pour génotype les mots de passes en eux-même. Le génotype est donc identique au phénotype.

Nous avons fait ce choix car nous n’avons pas trouvé de génotype qui serait différent du phénotype tout en étant efficace.

Nous avions également testé de définir que chaque lettre est un gène différent, néanmoins nous n'avons pas remarqué que de différence de performances. En particulier, nous avions implémenté ce système en testant sur la « nouvelle blackbox  (nous avons remarqué que certains fichiers blackbox étaient plus difficiles que d'autres pour notre algorithme. En particulier le fichier sur lequel nous avons testé la version un gène = une lettre nous a semblé plus difficile que le fichier que nous avons utilisé pour le reste du TP).

Une de nos idées étaient d’agrémenter le génotype de méta données. Par exemple, une de nos implémentations étaient de donner au mot une plage de lettres qui serait marquées comme « protégées » avec une mutation qui changeait des lettres non protégées dans une zone (par exemple BBBBAAA avec une protection sur les 4 premières lettres pouvait devenir BBBBCZA si une mutation se produisait sur le premier A). Cette implémentation donnait néanmoins de moins bons résultats que la version de base.

Un autre idée fût d’attribuer aux mots une liste de poids lors d’un crossing over, en donnant un poids de 0,2 pour les lettres dont les deux individus d’origines ont la même, et un poids de 0,6 pour les lettres dont les deux individus d’origines ont des lettres différentes. L'idée étant ensuite de protéger les lettres ayant un poids faible des mutations. Cette implémentation compliquait néanmoins grandement le code, pour un apport de performances faibles sur ce que nous avons testé.

Finalement, nous avons décidé d'ajouter comme métadonnées aux mots leur âge. L'idée étant que lorsqu'un mot est suffisament vieux (par exemple 20 générations sans mutation), nous avons atteint un minimum local. A ce moment, nous supprimons tous les autres mots ayant un moins bon

score afin de générer une nouvelle population. Le but de cette démarche est d'améliorer la diversité des mots.

## Sélection

Nous effectuons de l'élitisme pour la principale raison que expérimentalement cela semble être le plus performant pour converger rapidement vers un score élevé.

Lors de la sélection, nous utilisons un score qui est égal à l'opposé du rang (le 1er a un score de 100, le second un score de 99 …). Nous avions à l'origine choisit un poids égal au score, mais le lissage trop important du score à partir de 0,75 semble nuire fortement à l'amélioration du score.

L'élitisme que nous avons implémenté se décline en deux volets :

- Le premier est un processus de sélection naturelle, consistant par défaut à éliminer tous les individus sauf les 20 meilleurs
- Le second consiste à prendre dans l'ancienne population les X meilleurs mots (avec X = 8 par défaut) et de les ajouter à la nouvelle population sans faire de mutation. La motivation est ici de garder les meilleurs mots afin de ne pas perdre notre progression dans le score.

Les autres individus sont issus de croisements entre des mots différents ayant passé la sélection naturelle.

## Mutation

A chaque itération, nous n'appliquons une mutation sur les mots issus d'un croisement (à l'origine, nous avions défini un taux de mutation sur un mot entier inférieur à 1, mais expérimentalement, les meilleurs résultats sont lorsque l'on force la mutation des mots).

Les mutations que nous avons défini sont les suivantes :

| Nom | Description |
|---|---|
| Ajout / Suppression de lettres | Ajoute ou supprime une lettre à une position aléatoire |
| Inversion de lettres | Inverse deux lettres. |
| Changement de lettre | Change une lettre, soit pour une lettre aléatoire, soit pour une lettre proche dans l'alphabet |

Chaque mutation (ajout, suppression de lettres, inversion de lettres consécutives, inversion de lettres proches dans le mot (distance maximale de 5), changement de lettre aléatoire et changement de lettre proche dans l'alphabet) est équiprobable

Initialement, nous avions défini des poids différents, et nous changions ces poids lorsque le score du meilleur individu dépassait un seuil (0,92). Bien que ce procédé semblait bien marcher

pour résoudre la recherche d'un mot en particulier, il n'est pas générique (le réglage fait pour un mot de passe risque de ne pas marcher pour un autre mot de passe) et est

L'ajout et la suppression de lettres sont des opérations naturelles pour ce problème car il faut être capable d'explorer toutes les tailles de mots enter 12 et 18 lettres.

L'inversion de lettres vient de l'observation que souvent en fin d'exploration, la solution peut être trouvée en inversant deux ou trois lettres. De plus souvent ces lettres sont assez proches. Les différentes versions ont été faites pour rechercher un équilibre entre nombre d'échanges possibles (réduction de l'espace de recherche) et nécessité de faire des échanges plus ou moins loin (ne pas exclure des possibilités que l'on aurait aimé avoir).

Le changement de lettre peut être vu comme un raccourcis de la suppression puis l'ajout de caractère. C'est une opération qui semble également naturelle de base. La version en fin de mot vient de l'observation que souvent, le début du mot est relativement stable entre différentes exécutions.

Une autre mutation que nous avions implémenté et le décalage, consistant à déclarer toutes les lettres d'un cran vers la droite. Nous l'avons supprimé car elle ne nous semblait pas utile.

## Cross-over

Pour réaliser un cross-over, on se contente de prendre le début du premier mot et la fin du second mot.

La motivation principale est qu'une version naïve de la fonction de score serait que chaque lettre bien placée vaut 1/taille du mot et que le reste vaut 0 (le score serait la somme du score de chaque position de lettre). Si on a un mot dont le début est correct et un mot donc la fin est correcte, alors ce type de cross-over permet d'obtenir directement la solution.

Dans la pratique, cette fonction de cross-over semble marcher plutôt bien pour notre génotype, d'où le fait que nous l'avons conservé.

## Elites dégénérées

Enfin, afin de combattre les minimums locaux en fin d'exécution de l'algorithme, nous avons défini un mécanisme d'élites dégénérées : A chaque génération, nous prenons certaines élites, que nous clonons et faisons muter plusieurs fois (un nombre de fois égal à 40% de la taille du mot).

L'objectif de ce procédé est de forcer la production de plusieurs mutations, qui permettront de trouver la bonne mutation permettant de sortir du plateau final afin de trouver le bon mot de passe.

## Valeur des hyper paramètres

Les hyper paramètres que nous avons choisis par défaut sont les suivants :

| Paramètre | Valeur | Explication |
|---|---|---|
| Nombre d'individus | 100 | La valeur de 100 a été choisie par défaut car il s'agit de la valeur la plus souvent mentionnée en TP. |
| Sélection naturelle | 20 | Permet de filtrer rapidement les « mauvaises » mutations |
| Elites préservées | 8 | Le nombre d'élites préservées n'a pas l'air important, tant qu'il existe et est faible |
| Nombre d'élites dégénérées | 10<br>40% de mutations | Permet de forcer 60 nouvelles mutations par générations sur les meilleurs candidats |
| Âge des individus provoquant la mort de tous les moins bons | 50 | Nous avons remarqué que souvent, l'algorithme se stabilisait vers la 70eme génération, et pouvait ensuite prendre des milliers de générations avant de trouver un meilleur candidat. 50 nous semble être un bon compromis pour ne pas recommencer trop vite. |

## Ancienne stratégie pour sortir des plateaux : les sous-populations

Une stratégie que nous avions mis en place pour sortir des plateaux était la conception de sous-populations.

Nous commencions par lancer des algorithmes génétiques afin de trouver des candidats ayant un score dépassant 0,90. Une fois trouvé, nous gardions en mémoire ces candidats et recommencions l'algorithme.

Une fois 10 candidats trouvés de cette manière, nous lançions un ultime algorithme génétique, avec une population constituée de ces 10 « finalistes » afin de trouver le mot de passe.

Pour limiter le nombre de génération, nous avions également fixé le nombre maximum de générations des premières populations à 50 : au-delà de ce seuil, tous les individus étaient abandonnés et nous relancions une nouvelle exécution.

La motivation est ici de générer des individus variés ayant un bon score afin que lors de la dernière population, les cross over convergent rapidement vers une solution.

Le défaut de cette approche est néanmoins le nombre de générations utilisées, qui cumulé entre toutes les populations créées dépassait souvent le millier

# Etude des hyper-paramètres

Nous étudions ici la performance de notre algorithme en jouant sur les différents hyper-paramètres.

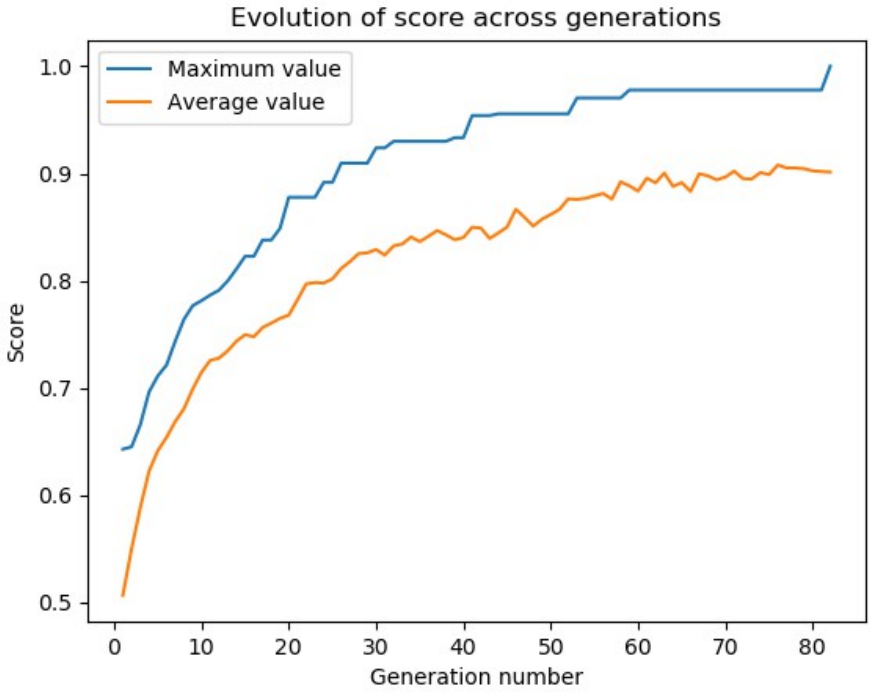
Pour rappel, les particularités de notre algorithmes sont :

- La présence d'une sélection naturelle filtrant les candidats, puis un certain nombre d'élites sont conservées telles quelles
- La présence d'élites dégénérées, des individus qui sont des clones des élites ayant subi un fort nombre de mutations
- Les autres individus sont issus de cross-over avec une mutation obligatoire
- Il est possible qu'à certaines étapes, un individu ayant un age précis décide de tuer tous les individus moins bon que lui, forçant une nouvelle diversité.

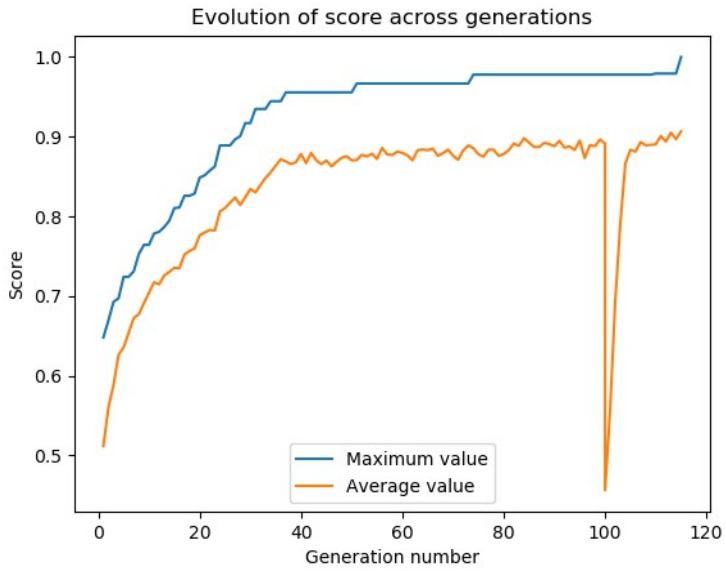
Nous étudions l'influence des hyper-paramètres en terme d'individus générés, car il nous semble que c'est la métrique la plus intéressante. En effet, la génération d'individus est la mesure qui nous semble la plus proportionnelle au temps d'exécution (qui est la mesure la plus importante pour un utilisateur quelconque mais qui est imprécise, par exemple à cause des interruptions ou du garbage collector de Python).

Les mesures sont faites sur plusieurs itérations, en prenant le pus souvent la moyenne de 10 exécutions.

## Performance générale

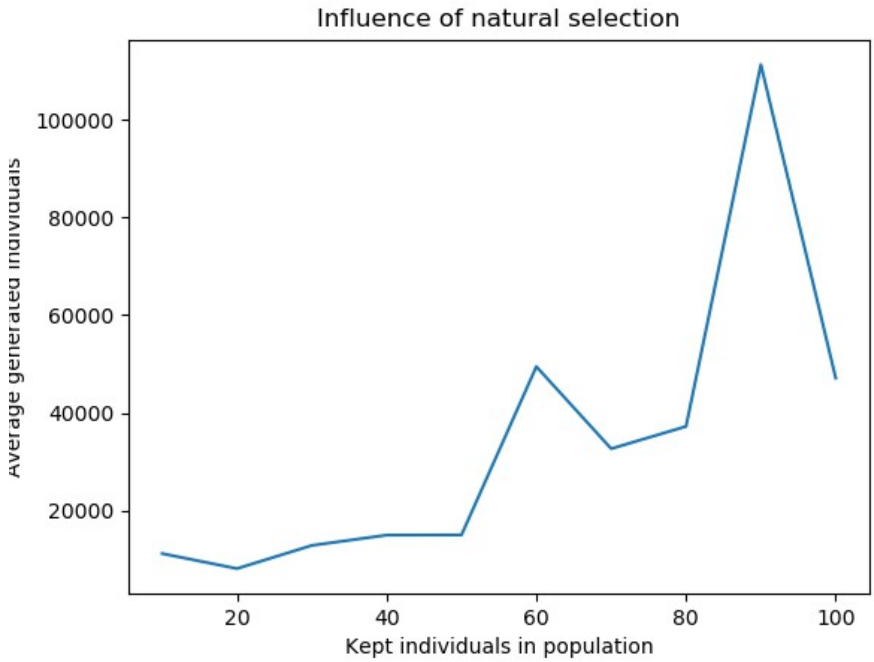


Cette première courbe expose la croissance rapide du score trouvé pour les mots de passe générés. L'autre aspect mis en valuer est la présence d'un plateau, visible ici entre les générations 60 et 80.



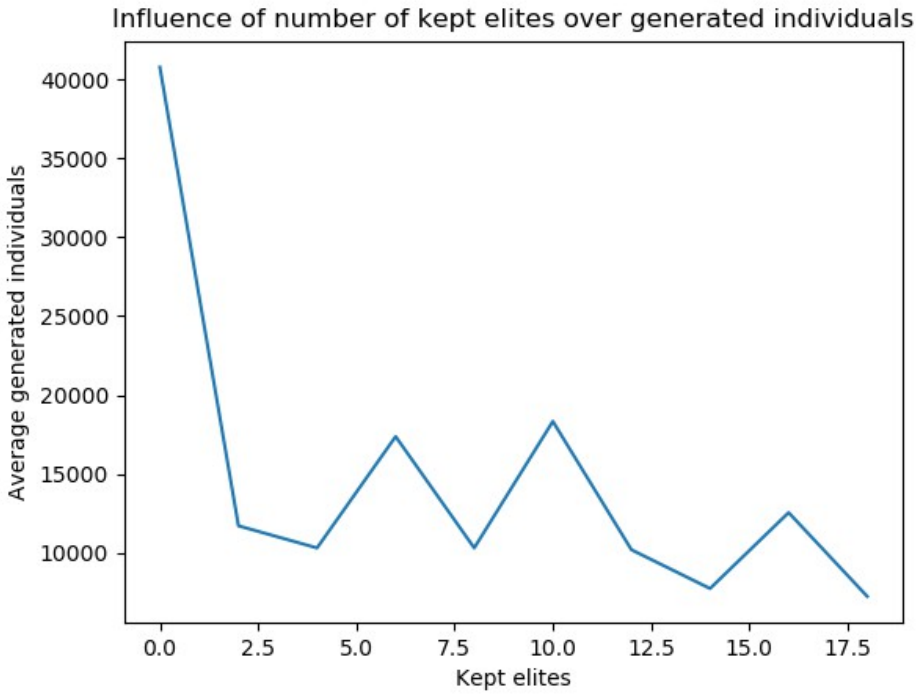

Parfois, on retrouve une courbe avec une chute de la moyenne du score. Cela est dû au mécanisme d'exécution de la population. On peut voir que par rapport à l'exécution totale, la reconstruction d'une nouvelle population est rapide par rapport au nombre de générations (la conception de l'algorithme garantissant que les individus de cette nouvelle diversité ne sont pas influencés dans leur évolution par les membres de l'ancienne population).

## Influence de la sélection naturelle



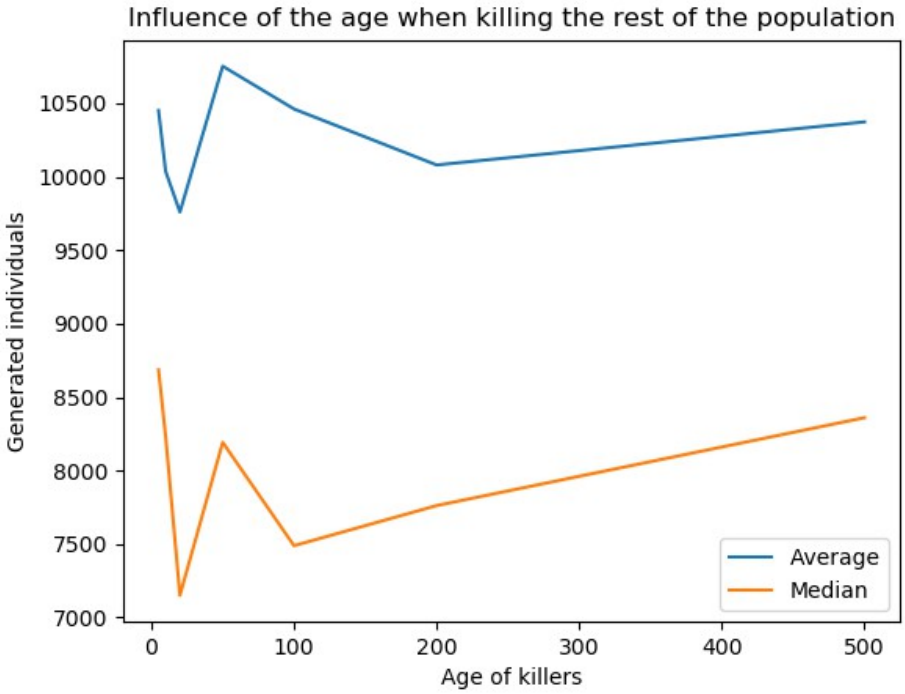

Nous pouvons voir que l'impact du nombre d'individus gardés est faible, tant qu'il est relativement faible. 20 individus semble être l'idéal.

## Influence de la taille de l'élite




A cause du fort taux de mutation, on peut voir que la conservation d'un certain nombre d'élites non mutées est quelque chose d'important. Si la courbe semble légèrement décroite après 14 individus, on remarque néanmoins que l'on reste de manière stable vers 10000 individus.

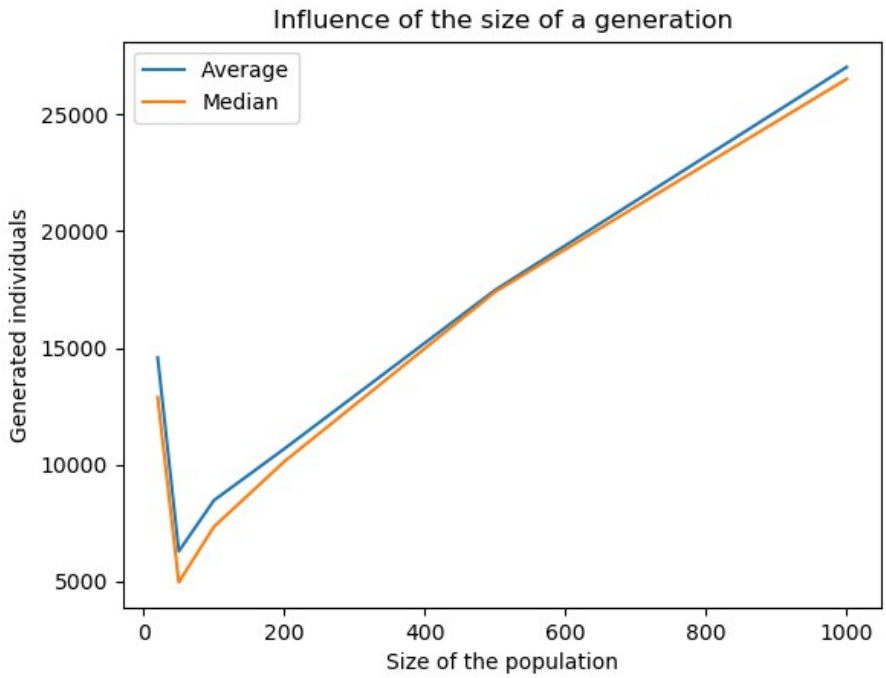
## Influence de l'âge auquel les individus tuent les autres membres de la population




Nous étudions ici l'impact de l'âge d'un individu qui n'a pas muté ni subi de crossover (un de ces deux scénarios faisant recommencer l'âge à 0) tue les individus moins bons sur le nombre d'individus générés au total.

Si nous avions implémenté à l'origine ce mécanisme pour éviter les minimums locaux, nous ne voyons pas d'impact particulier sur cette courbe. Dans la mesure où intuitivement, nous avons eu l'impression que l'instauration de ce mécanisme a réduit le nombre de générations des exécutions longues, nous pensons que nous aurions dû comparer le nombre maximal de générations sur des exécutions avec et sans élimination de la population.

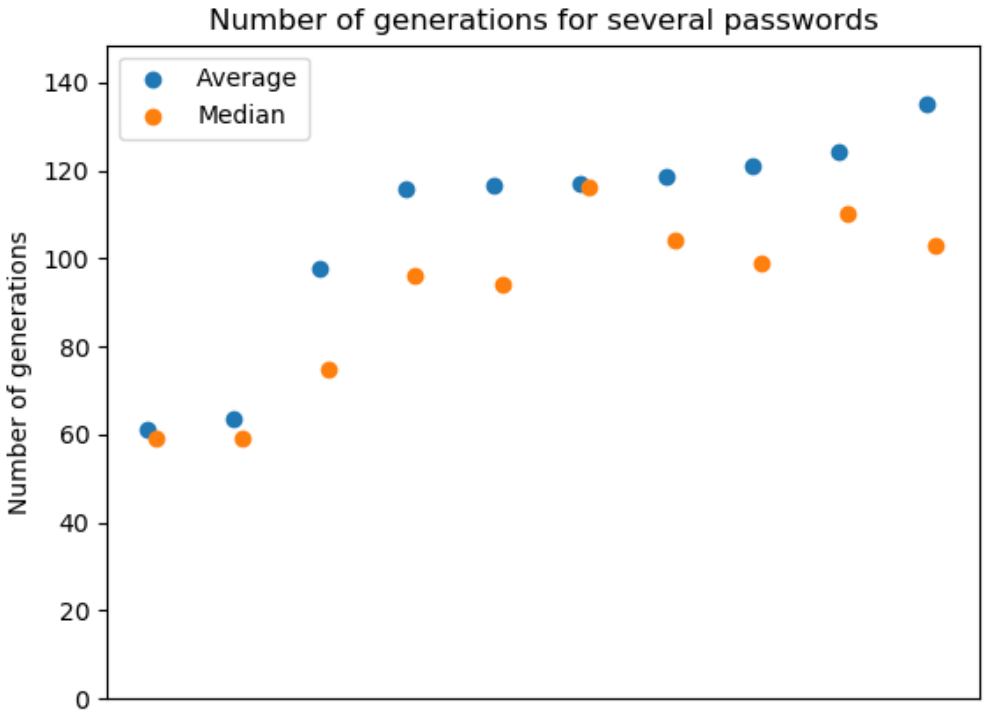
## Impact de la taille de la population

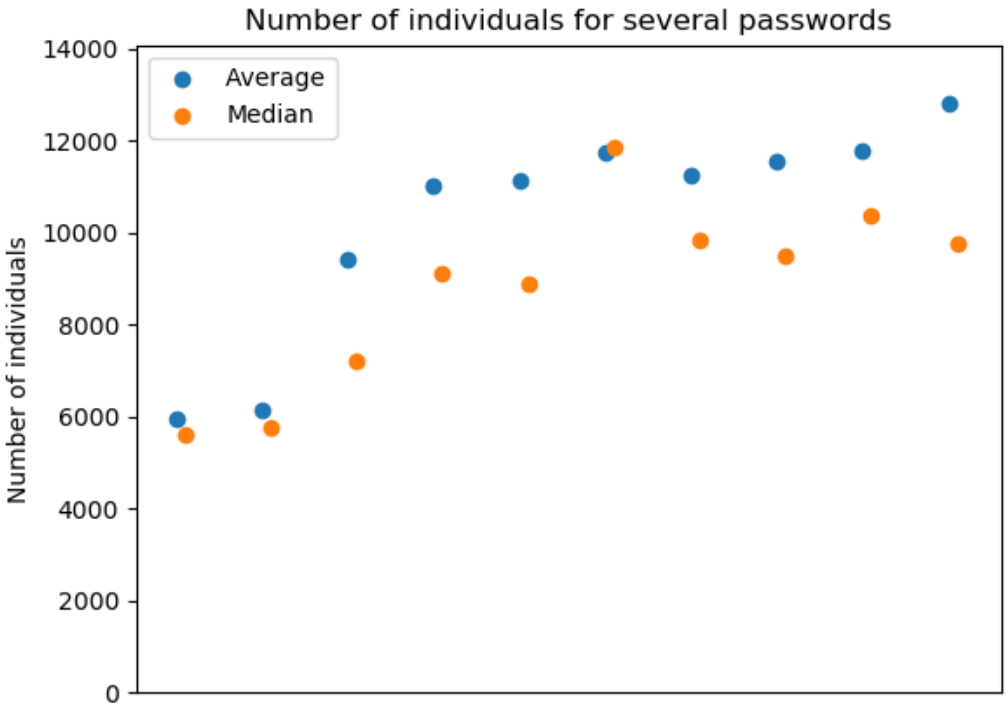


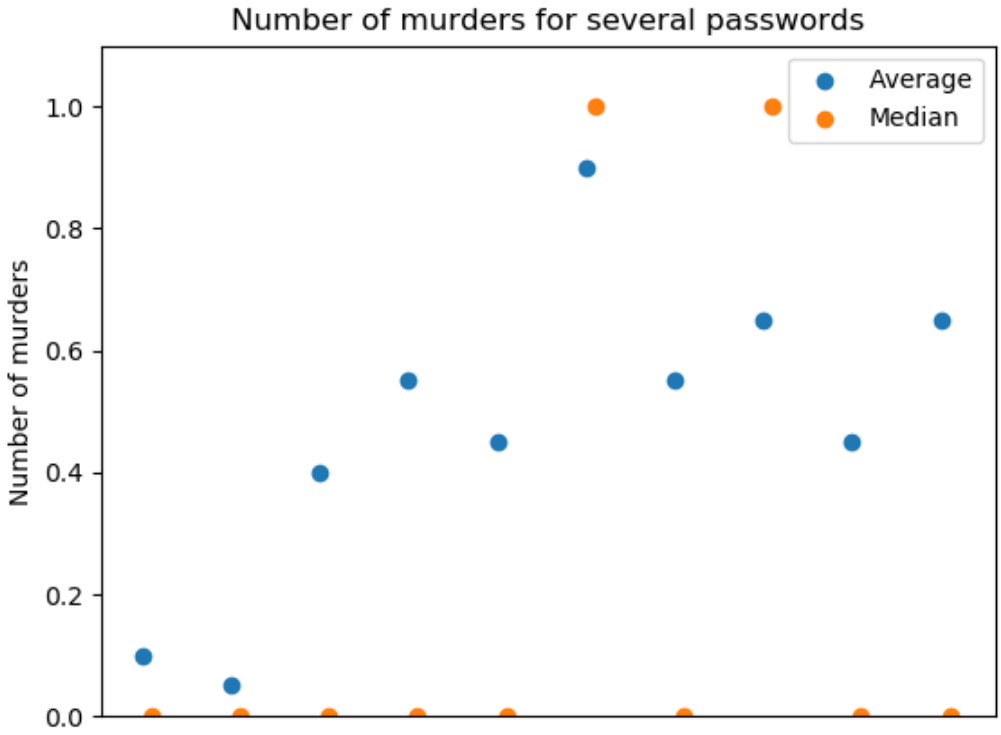

Avec les paramètres testés, nous voyons que la taille de population la plus efficace est de 50, et que le nombre d'individus générés croit ensuite proportionnellement avec la taille de la population. A noter que pour ce test, nous avons défini un taux de sélection naturelle (20%), nombre d'élites gardées (10%) et un taux d'élites dégénérées (5%) proportionnel à la taille de la population. Cela n'a pas empéché la génération d'individus que l'on pourrait considérer superfluxs, car ne semblant pas contribuer à la construction d'une solution.

## Efficacité sur plusieurs mots de passes

Enfin, nous avons testé l'efficacité de notre méthode sur 10 mots de passes différents, dont nous avons trié le nombre de générations moyennes et affiché les résultats sur les graphiques suivant (chaque point correspond à un mot de passe). Ici, nous avons fait le choix d'afficher la totalité des données qui nous semblaient pertinentes pour l'analyse : le nombre d'individus créés, le nombre de générations et le nombre de génocides de la population.




Number of individuals for several passwords
Average
Median
Number of individuals
14000
12000
10000
8000
6000
4000
2000
0


Number of murders for several passwords
Average
Median
Number of murders
1.0
0.8
0.6
0.4
0.2
0.0

Nous pouvons voir que de manière générale, les performances de notre algorithme sur les différents mots de passes testées sont très constantes : la moyenne et la médiane sont proches, et le nombre de générations / individus / génocides sont également proches pour différents mots de passes.

La méthode que nous avons implémenté est donc suffisament générique pour trouver de manière efficace n'importe quel mot de passe issus de la blackbox que nous avons testé.

### Remarque sur la difficulté de la « nouvelle blackbox »

Nous avions téléchargé le fichier blackbox pour python 3.7 au début du TP et nous avons remarqué que ce fichier est beaucoup plus simple à résoudre que le fichier backbox actuellement disponible pour python 3.6. Nous qualifions cette blackbox plus complexe à résoudre de « nouvelle blackbox » .

Avec le fichier de la version 3.6, nous arrivons régulièrement à des minimum locaux avec un score de 0,833, avec par exemple le meilleur candidat étant JXJOLGFO8PV9WO2 et le mot à trouver étant XJJG8OLFOPO92VW.

Nous avons préféré revenir au fichier d'origine car notre algorithme avait été codé pour ce fichier. Une approche que nous n'avons pas testé serait de créer une mutation « permuter un groupe de lettres » qui permuttrait aléatoirement un groupe de 3 à 6 lettres dans l'espoir de sortir du minimum local impliqué par le fait de ne permuter les lettres que par groupe de deux.

## Conclusion

Notre algorithme est très efficace pour résoudre les problèmes de la blackbox 3.7 mais inutile pour les problèmes de la blackbox 3.6. Ce fait montre qu'il n'y a pas de solution générique pour résoudre un problème, et qu'un algorithme génétique peut ne pas fonctionner si les fonctions de mutations prévues ne sont pas adaptées, même si il est très efficace pour un autre problème similaire.