IA5 - Intelligence Bio-Inspiré - Algorithmes génétiques
Rendu de Travaux Pratiques

Alexis Pister et Raphael Teitgen

Encadrant : Alexandre Galdeano

December 2018

1 Introduction

Le but de ce TP est de retrouver un mot de passe inconnu de 12 caractères de la forme ([0-9A-Z_]{12}) à l'aide d'un algorithme génétique. Le principe est de considérer une population de mot de passes considérés comme des individus, que nous allons faire évoluer à l'aide de croisements et de mutations. A chaque génération, certains des mots de passe seront sélectionnés à l'aide de la fitness calculée de chaque individu et d'aléatoire. La fitness est calculée à l'aide d'une formule inconnue, qu'il sera intéressant d'étudier une fois le mot de passe trouvé.

2 Pseudo-code

Dans les fonctions expliquées ci-dessous, n désigne le nombre d'individus, *tailleIndividu* le nombre d'éléments constituant un phénotype (ici 12) et *caracteres_poss* une liste contenant les briques élémentaires possibles pour le phenotype, ici ([0-9A-Z_])

Le génotype des individus a été encodé en convertissant chaque caractère en son indice dans la liste des 37 possibles. Par exemple 'B' sera converti en 1, étant en 2ème position dans la liste (celle-ci commence à 0). Cet encodage a été choisi afin que les mutations changent les caractères en de caractères relativement proches, faisant varier les individus petit à petit.

fonction phenotypes_to_genotypes(*phenotypes***):**
    Initialiser *genotypes* une liste vide
    Pour i de 1 à n
        Pour j de 1 à *tailleIndividu*
            genotypes[*i*][*j*] prend comme valeur l'indice de *phenotypes*[*i*][*j*] dans *caracteres_poss*
    Retouner *genotypes*

fonction genotypes_to_phenotypes(*genotypes***):**
    Initialiser *phenotypes* une liste vide
    Pour i de 1 à n
        Pour j de 1 à *tailleIndividu*
            phenotypes[*i*][*j*] prend comme valeur *caracteres_poss*[*genotypes*[*i*][*j*]]
    Retouner *phenotypes*

Deux techniques de croisement ont été implementées, chacune ayant une chance sur 2 d'avoir lieu à chaque croisement. La première consiste au crossing over classique ou les génome des 2 enfants résultent d'une coupure à une position aléatoire des 2 parents et d'une permutation des 2 parties. La deuxième revient en une distribution aléatoire de chaque position des 2 parents vers les 2 enfants. La 2ème techniques permet de mieux brasser l'espace des génotypes possibles, mais les 2 techniques ont été implémentés pour augmenter la diversité des enfants générés.

fonction croisement(*individuA***,** *individuB***):**
    Tirer r aleatoirement entre 0 et 1

Initialiser $enfantA$ et $enfantB$ 2 listes vides
Si r < 0.5
    Tirer un entier aleatoire k entre 1 et n
    Permuter les caractères 1 à k de $individuA$ avec les caractères k+1 à n de $individuB$ et mettre les resultats dans $enfantA$ et $enfantB$
Sinon
    Pour i de 1 à $tailleIndividu$
        Tirer p aleatoirement entre 0 et 1
        Si p < 0.5
            $enfantA[i] = IndividuA[i]$
            $enfantB[i] = IndividuB[i]$
        Sinon
            $enfantA[i] = IndividuB[i]$
            $enfantB[i] = IndividuA[i]$
Retourner $enfantA$, $enfantB$

Pour la sélection, les individus sont choisis aléatoirement, parmi les plus mauvais et à l'aide d'une sélection par roulette exponentielle par rang avec des proportions paramétrables. La sélection par la roulette permet de sélectionner les meilleurs individus avec une plus forte probabilité à chaque pas de temps afin de converger vers le meilleurs individu possible. La technique par le rang exponentiel est utilisé pour favoriser la reproduction d'un individu qui sort du lot si l'algorithme stagne à un moment donné. Une proportion d'individu est choisi parmi les plus mauvais génomes et aléatoirement afin de favoriser la diversité et ne pas rester dans un minimum local.

fonction selection($genotypes, fitnesses, c, freqRand, freqBad, probCO$**):**
Donner un rang r a chaque genotype par rapport à sa fitness
Creer une liste de poids pour chaque individu avec $w(r) = \frac{c-1}{c^n-1}c^{n-r}$
Selectionner une proportion $freqBad$ des plus mauvais genotypes
Selectionner une proportion $freqRand$ de genoypes aléatoirement
Selectionner le reste des genotypes avec des probabilitées correspondantes à leurs poids w_i (avec remise) pour obtenir au total n individus
Mettre les genotypes selectionnés dans une liste $genomeSelected$
Initialiser $newGenomes$ comme liste vide
Tant que i $<$ n
    Tirer r aleatroiement entre 0 et 1
    Si r $<$ $probCO$
        $enfantA, enfantB =$ croisement$(genomeSelected[i], genomeSelected[i+1])$
    Sinon
        $enfantA, enfantB = genomeSelected[i], genomeSelected[i+1]$
    Rajouter $enfantA$ et $enfantB$ à $newGenomes$
    $i = i + 2$
Retourner $newGenomes$

La mutation d'un gène consiste à lui incrémenter un entier tiré d'une loi normal de moyenne de la valeur du gène étudié et d'écart type 10. Cela a pour but de transformer les gène vers des gènes relativement proches. De plus, il y a une chance de 5% que la mutation consiste en une permutation du gène avec le gène adjacent, afin d'augmenter la diversité des génomes obtenus après mutation.

fonction mutation($genotype, pMutation$**):**
Pour i de 1 à $tailleIndividu$
    Tirer p aléatoirement entre 0 et 1
    Si p $< pMutation$
        Tirer r aleatoirement entre 0 et 1
        Si r < 0.95
            $genotype[i] = genotype[i] + int(N(0,10))\%length(caracteres\_poss))$

Sinon
Permuter $genotype[i]$ et $genotype[i+1]$
Retourner $genotype$

3 Résultats

Le mot de passe a finalement été trouvé avec les paramètres suivants au bout d'environ 40000 générations de 400 individus.

- $pMutation = 0.15$
- $probCO = 0.75$
- $c = 0.95$
- $freqBad = 0.15$
- $freqRand = 0.05$

Il s'agit de **_CH3NI_P4N__**.
Des probabilités de mutation et de crossing-over élevées ont été choisies afin de brasser beaucoup de diversité à chaque génération. Le paramètre $c = 0.95$ lors de la sélection par la roulette exponentielle permet de favoriser les meilleurs individus mais pas trop non plus afin que certains individus moyens soient choisis de temps en temps. Des proportions $freqBad = 0.15$ et $freqRand = 0.05$ ont été choisies afin de conserver une diversité dans la population. La fig. 3 présente l'évolution de la fitness moyenne et maximum à travers les générations. On peut voir que la fitness maximum stagne tout le long de la simulation à 0.63715 avant d'atteindre 1, ce qui doit être un minimum local. Cependant, on peut voir que la fitness moyenne varie beaucoup entre 0.35 et 0.5 le long de la simulation, signe d'une bonne diversité. C'est cela qui a permis de trouver le bon mot de passe avec le temps.

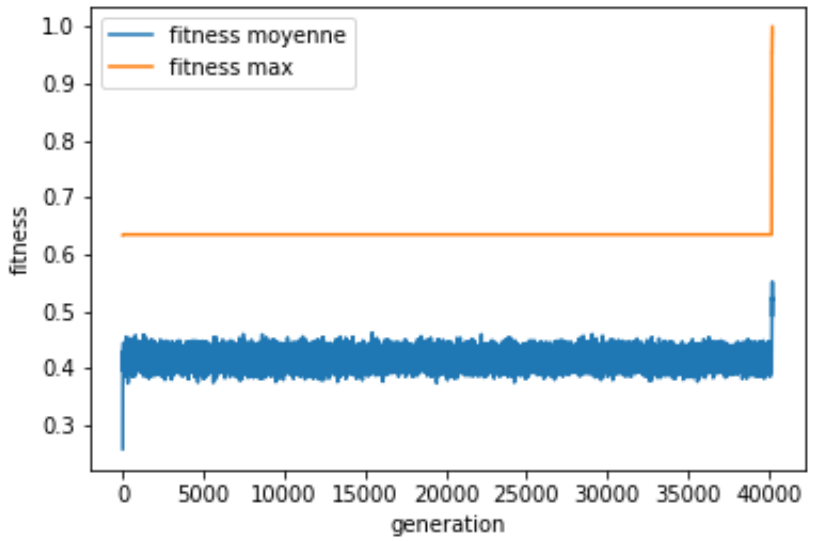

Figure 1: Évolution de la fitness moyenne et maximum au cours des générations

4 Bonus

Pour trouver la courbe de la fonction fitness, on a cherché tout d'abord à déterminer l'apport d'une lettre dans un un mot, c'est à dire en comparaison avec le mot de passe à trouver quel est l'impact d'une lettre sur le score de la fitness.

On a donc regardé ce qu'implique la distance entre une lettre correcte et une autre lettre de l'alphabet proposé dans le tp, c'est à dire ([0-9A-Z_]). Tous les autres caractères du mot de passe étaient bon. Sur le graphique suivant on peut observer que plus une lettre proposée est proche de la lettre du mot de passe à déterminer, à emplacement respectif, et plus le score est élevé. A l'inverse, on observe que plus la distance est éloignée dans l'alphabet et plus le score est mauvais. On a ajouté une lettre au hasard pour comparer le pire/meilleur score possible.

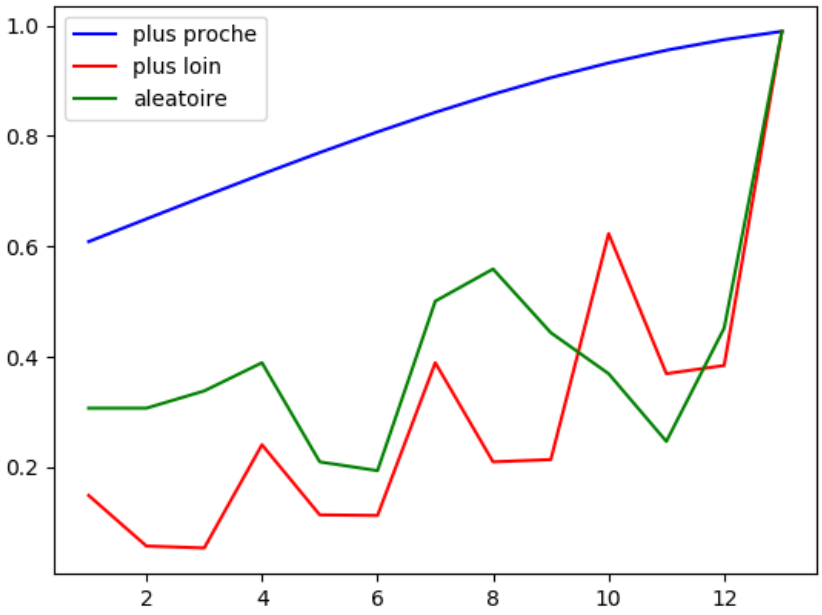

Figure 2: Valeur de la fitness du mot en fonction de l'écart d'un seul caractère au bon caractère

Cela nous confirme donc l'intérêt de la distance entre les lettres. On va donc créer la droite de l'évolution de la courbe. L'alphabet possède 37 caractères, donc le pire score de fitness est atteint à une distance de 18 pour chaque caractère. Sachant qu'il y a 12 lettres dans notre mot de passe, on doit générer 12*18 mots différents. Ici on va faire varier les mots du meilleur au pire, dans notre cas le pire mot de passe possible est IUZM4_I6N4II, c'est lui qui donne la moins bonne fitness possible. On part donc à l'indice 0 et on s'éloigne du mot de passe original.
On peut observer sur la courbe suivante l'évolution de la fitness en fonction du nombre de modification effectuées. Cette fonction nous montre quelque chose d'intéressant sur la fonction de fitness : le calcul de la distance pour la fitness entre le caractère réel et un caractère testé n'est pas la même en fonction de l'indice. En effet on peut observer 3 cas : le premier montre que s'éloigné du caractère original fait chuté le score de fitness, un second cas montre qu'au contraire ça semble l'améliorer (en dehors d'avoir le réel caractère qui lui améliore toujours la fitness) et enfin une variation, le troisième cas fait chuté la fitness plus on s'éloigne et à partir d'une certaine distance il va l'améliorer.
On peut donc en conclure trois choses, tout d'abord il y a une relation forte entre distance d'un mot et mot de passe réel et la fitness, ensuite suivant l'indice la distance agit différemment sur la fitness. Enfin il est très difficile, voir impossible, de retrouver la structure exacte du calcul de la fonction de fitness à cause de ces variations.

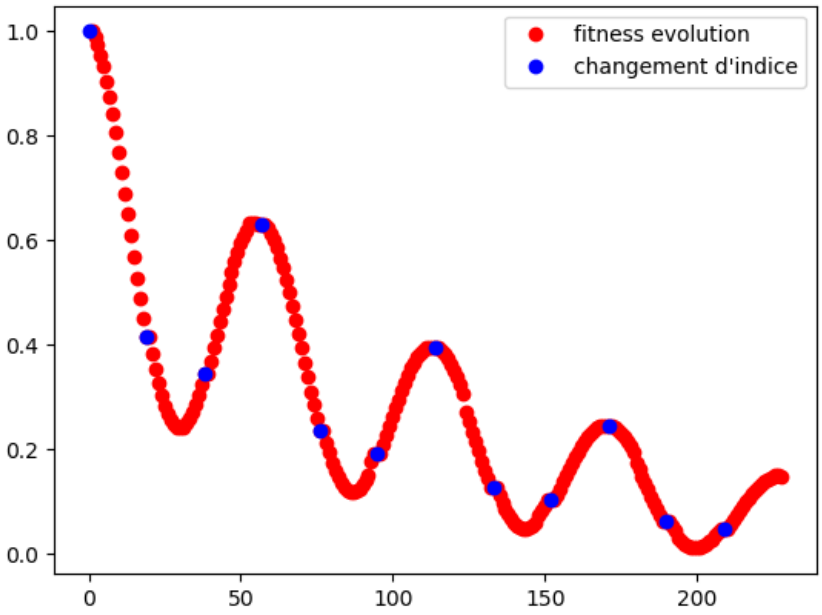

Figure 3: Évolution de la fitness d'un mot en fonction de sa distance au mot de passe original