Programmation C avancée
TP 5

L'objectif de ce TP est de casser un code binaire inconnu avec une attaque par programmation génétique.

Le "coffre" à ouvrir vous est fournis dans un fichier code.o qui contient les deux fonctions: `fitness_key`, et `enter_the_matrix` déjà compilées et que vous devez donc linker à votre code. Et un fichier code.h qui contient les définitions ci dessous.

```
1 #define NB_OCT 32
2
3 /* System free C type for binary keys */
4 typedef struct{
5   uint8_t values[NB_OCT];
6 }Bitkey;
```

C'est finalement une clé sur 256 bits mais qui donne pas mal de différentes possibilités pour ceux qui voudraient tout tester...

```
1 /* Return a mesure of the quality of the given key 'k'. However, the
2    returned value is unsafe. The only warranty is that a perfect key
3    has for fitness 100.0000 */
4 float fitness_key(Bitkey* k);
5
6 /* This function allows you to enter the matrix only if you have the
7    perfect key. Are you ready for this experience ? */
8 void enter_the_matrix(Bitkey* k);
```

À partir de là, tous les coups sont permis, sauf l'utilisation d'un décompilateur. C'est triché et en plus, cela dénoterait un grave manque de classe de votre part.

Méthode 1 :

C'est la méthode : "Moi je maîtrise la programmation génétique. Je manipule une population avec croisements, mutations et je casse le code en deux deux".

C'est possible mais pas si facile que ça... La fonction de `fitness` peut se tromper et donc vous induire en erreur. Mais même si elle se trompe, statistiquement, sur un grand nombre de tests, elle donne bien une `fitness` précise. Si cette indication vous suffit et que vous avez une bonne expérience de la programmation génétique, n'hésitez pas !

Attention, c'est pas si facile que ça de faire un algorithme convergeant. Il faut finement programmer les fonctions de mutation et de croisement pour espérer percer le code en temps raisonnable.

Méthode 2 :

C'est la méthode : "Je vais écouter les conseils du prof, ça peut aider".

L'attaque consiste, à partir de clés aléatoires, de les classer selon leur `fitness` et d'accoupler les meilleurs pour obtenir des enfants de qualité encore meilleure.

Conseil 1 : accouplement à trois

À deux, c'est pas mal mais à trois c'est tellement mieux. Pour accélérer la convergence, on accouple les clés par trois. Lorsque deux clés sont bonnes mais que pour un même bit, elles ont deux valeurs différentes, c'est dur de choisir pour l'enfant (on a une chance sur deux). Avec trois clés, on accouple bit par bit comme il suit : on fait la somme des bits des trois parents. Si la somme fait 0 ou 1, on a de fortes chances que, parmi ces bonnes clés, juste une seule se trompe et donc on met un `0` pour l'enfant. Si la somme fait 2 ou 3, comme encore une fois, probablement une seule se trompe, on met un 1 pour l'enfant. On évite ainsi l'indécision d'un pile ou face et on accélère grandement la convergence.

Cela donne donc pour une clé binaire sur un char :

	1	1	0	1	0	0	1	0	`parent 1`
	1	0	1	1	0	1	1	0	`parent 2`
&&&	1	0	1	1	0	0	1	1	`parent 3`
	1	0	1	1	0	0	1	0	`enfant`

Cet accouplement à trois réalise aussi l'intuition de garder les meilleures parties de ces trois parents.

Conseil 2 : utiliser un maximum d'individus aléatoires

Les attaques classiques de programmation génétique souffrent ici à cause de la faiblesse de précision de la fonction de `fitness`. Lors d'une petite mutation (1 seul bit), même si cette dernière est bonne, la `fitness` peut rendre une quantité moins importante. Pour éviter ce problème, il faut absolument générer un grand nombre d'individus et garder les meilleurs, statistiquement, ça finira par converger. Étant donné la faiblesse de sémantique de ce problème (toutes les clés, à priori, sont bonnes), la dégénérescence du génome est un réel problème.

Conseil 3 : une manière de procéder

Il est possible de produire un programme en 40 lignes cassant ce code en quelques secondes. Pour cela, il faudra une fonction d'accouplement, une fonction générant une clé aléatoire et une fonction qui trie un tableau de clé selon la `fitness`.

Enfin une fonction intelligente dont voici le pseudo-code :

```
generate_cle_generation(Bitkey* B, entier d){
  si d = 0
    retourne une clé aléatoire dans B

  allocation d'un tableau de 8 clés
  génération de 8 clé de génération d-1 /* 10 appels récursifs */
  tri du tableau
  accouplement des 3 meilleurs clés et fabrication de l'enfant dans B
  libération mémoire du tableau
}
```

Une clé de génération 7 ouvre le coffre bien souvent !