

Projet de C++ :

Algorithme génétique pour la recherche de chaîne



17/05/2016

Maxime Aberton

Rafael CARTENET

Killian JAUBERT

Sommaire

Introduction.....	4
Présentation de l'algorithme génétique.....	5
Analyse des besoins.....	7
Jeu d'essai.....	7
Spécifications fonctionnelles.....	8
Vision du projet.....	8
Amorce de la solution en termes de composants.....	8
Fonctionnalités.....	8
Description du domaine.....	9
Interactions entre les composants (associations).....	9
Tests d'intégration.....	10
Implémentation et test de l'algorithme.....	11
Répartition des tâches.....	16
Conclusion.....	17

Introduction

Dans le cadre de notre projet C++ nous avons décidé de réaliser un algorithme génétique afin d'une part, se familiariser au langage et d'autre part, se renseigner et implémenter différentes méthodes d'algorithmes génétiques que l'on trouve aujourd'hui et illustrer leur performances.

Nous avons donc décidé d'appliquer ce type d'algorithme à un problème relativement simple : retrouver une chaîne de caractères (target) entrée au préalable.

L'application étant relativement élémentaire, nous essayerons de paramétrer les opérations (sélections, crossovers, ...) de la manière la plus optimale possible, afin d'obtenir la solution le plus rapidement.

Grâce à l'implémentation de différentes méthodes, nous pourrons comparer les différentes performances. Nous pouvons décrire les performances d'un algorithme génétique par :

- Son nombre de génération pour accéder à une solution proche de l'optimale
- Sa capacité à éviter les maximums locaux
- Son temps calcul, qui est certainement l'élément principal.

De fait, notre algorithme est composé de nombreux paramètres, définis dans le chapitre suivant. C'est en faisant varier ces paramètres que nous pourrons évaluer de la performance de notre algorithme selon ces différents critères.

Présentation de l'algorithme génétique

Dans le domaine de l'intelligence artificielle, un algorithme génétique est une heuristique de recherche qui s'inspire grandement du processus de sélection naturelle. Dans notre cas, nous allons l'utiliser pour trouver une solution à un problème de recherche, ici la recherche d'une chaîne *target*, qui sera de taille SIZE.

L'algorithme génétique appartient à la classe des algorithmes évolutionnistes, il a donc pour but de converger vers une solution optimale, en passant par des étapes de mutation, de sélection et de crossover, termes que nous expliciterons par la suite.

Comment fonctionne un tel algorithme ? Nous partons d'un génome, qui contiendra une chaîne de caractères complètement aléatoire à la base, ainsi qu'un nombre appelé *fitness*, qui correspondra au degré de ressemblance entre la chaîne générée aléatoirement et la chaîne *target* – la fitness que nous avons utiliser est la suivante : si les caractères au même endroit d'une chaîne sont identiques, augmenter la fitness de 1, sinon ne rien faire. On cherche alors soit à maximiser, soit à minimiser cette fitness – dans notre cas, on cherchera à la maximiser.

Seulement, si l'on se contente de générer aléatoirement des chaînes de caractères, il y a peu de chances que l'on tombe sur la chaîne *target*. Ainsi au lieu de générer un seul génome, on va générer une population de génomes.

A chaque génération, on va appliquer plusieurs étapes à notre population :

- Mutation : pour chaque génome, on va faire muter chaque lettre avec une probabilité MUTERATE. Si cette lettre doit être changée, on la remplace par une autre lettre aléatoire de la table ASCII.
- Crossover : Pour chaque paire de génome, on va décider avec une probabilité CROSSOVERRATE si l'on effectue un crossover entre ces 2 génomes. Si oui, alors on choisit n (avec $n < \text{SIZE}$, n aléatoire) caractères dans le la string du premier génome, et $\text{SIZE} - n$ dans le second.
- Sélection : On trie ici nos génomes par ordre de fitness, si bien que les meilleurs se retrouvent au début de la population. On en conserve un pourcentage ELITRATE intacts : c'est le principe d'élitisme, pour éviter que notre solution diminue d'une génération à l'autre, on conserve les meilleurs de la génération précédente sur la suivante. Ainsi, la fitness ne peut qu'augmenter d'une génération à l'autre.

Ensuite, on va choisir les génomes qui vont perdurer dans la génération suivante, parmi ceux qui n'ont pas été définis comme l'élite... Pour ce faire, on applique le processus de Russian Wheel Selection – on additionne les fitness de tous les génomes de notre population, ce qui nous donne un entier F. On choisit alors un entier dans l'intervalle $[0;F]$ et prendre le génome dont la fitness se situe à cet endroit dans l'intervalle. Ainsi, les génomes avec une grande fitness ont plus de chances d'être sélectionnés, tout en gardant une part d'aléatoire.

- Mise à jour du MUTERATE variable : Nous mentionnions tout à l'heure l'existence de maximum locaux. Cela est généralement dû à une stagnation de l'algorithme : celui-ci brasse des solutions, mais n'en trouve pas de meilleure. Pour y remédier, nous avons pensé à implémenter un MUTERATE variable.

Lors des premières générations, on souhaite que le MUTERATE soit faible, pour la simple raison qu'un MUTERATE trop élevé fait varier les génomes trop drastiquement, ce qui les empêche de trouver une solution raisonnable, et donc de converger à une vitesse acceptable. Donc, au début, on souhaite un MUTERATE faible. Mais, lorsque l'algorithme est très proche de la solution finale (qu'il lui manque 2 ou 3 lettres), on désire augmenter le MUTERATE pour que l'algorithme teste plus de possibilités pour les rares lettres manquantes.

Ainsi, on va tester la stagnation de l'algorithme : les génomes étant classés par ordre de fitness, on va faire une différence entre la fitness du premier génome et celle du génome médian. Si cette différence est inférieure à un seuil défini au préalable (nous avons pris la plupart du temps un seuil de 1), alors cela signifie qu'au moins la moitié de la population a la même fitness que le meilleur génome, et que donc il est temps d'augmenter le MUTERATE, en le multipliant par un coefficient par exemple.

Après toutes ces étapes, on peut enfin ressortir le génome avec la meilleure fitness de cette génération. Si celle-ci est égale à SIZE, alors on a trouvé la chaîne target. Sinon, on crée une nouvelle génération de génomes, et on réitère nos différentes étapes.

Analyse des besoins

Le principe du projet étant assez simple, implémenter un algorithme génétique répondant à un certain problème, l'application n'a qu'une seule fonctionnalité : résoudre le problème. Le plus intéressant est la paramétrisation de l'algorithme afin d'obtenir la solution optimale le plus rapidement possible. De ce fait, nous nous efforcerons dans ce projet de rendre un programme avec les meilleurs paramètres possible pour répondre au problème.

Nous proposons ainsi le jeu d'essai suivant, afin de pouvoir valider du bon fonctionnement de notre programme.

Jeu d'essai

Problème	Résultat
La chaîne de départ est vide	Avertissement
La chaîne d'arrivée est vide	Avertissement
Un des paramètres (nb de gène max, taille population, ...) est nul	Avertissement
Paramètres corrects	Nombre de génération pour atteindre la solution

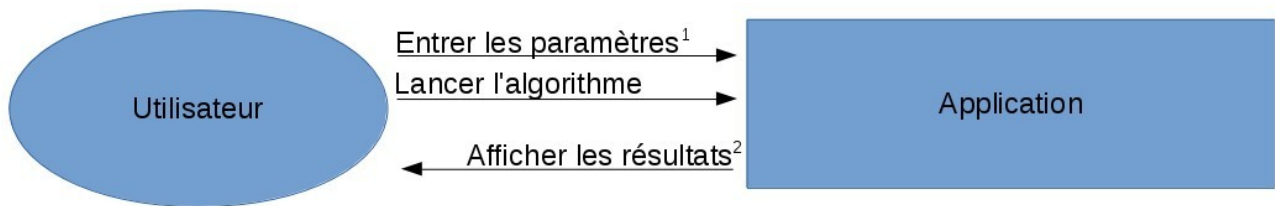
Spécifications fonctionnelles

Vision du projet

Le but du programme étant assez simple, il est donc logique son utilisation l'est aussi.

En effet, l'utilisateur pourra seulement entrer les différents paramètres de l'algorithme génétique ainsi que la chaîne à trouver et lancer l'algorithme. Il sera ensuite affiché à l'écran les résultats de l'algorithme. Dans un but pédagogique, nous afficherons toutes les étapes de l'algorithme afin de se rendre compte de l'efficacité de l'algorithme.

Le diagramme d'organisation métier se présente donc de la façon suivante :



1 : Chaîne initiale et chaîne finale

2 : Nombre de génération, temps de calcul, fitness génération par génération.

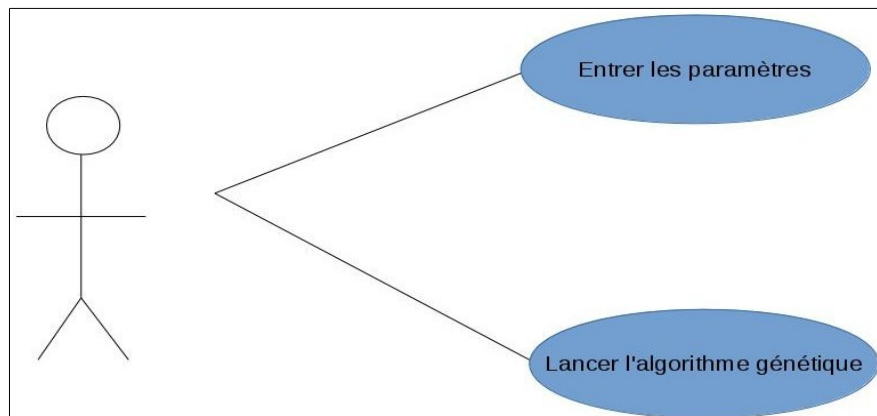
Amorce de la solution en termes de composants

Nous comptons décomposer notre programme en 2 modules :

- Une classe représentant un génome singulier
- Une classe permettant de représenter un ensemble de génome pour pouvoir lui appliquer l'algorithme génétique.

Fonctionnalités

L'étude des besoins de notre projet nous a donc amené à établir le diagramme des cas d'utilisation suivant :

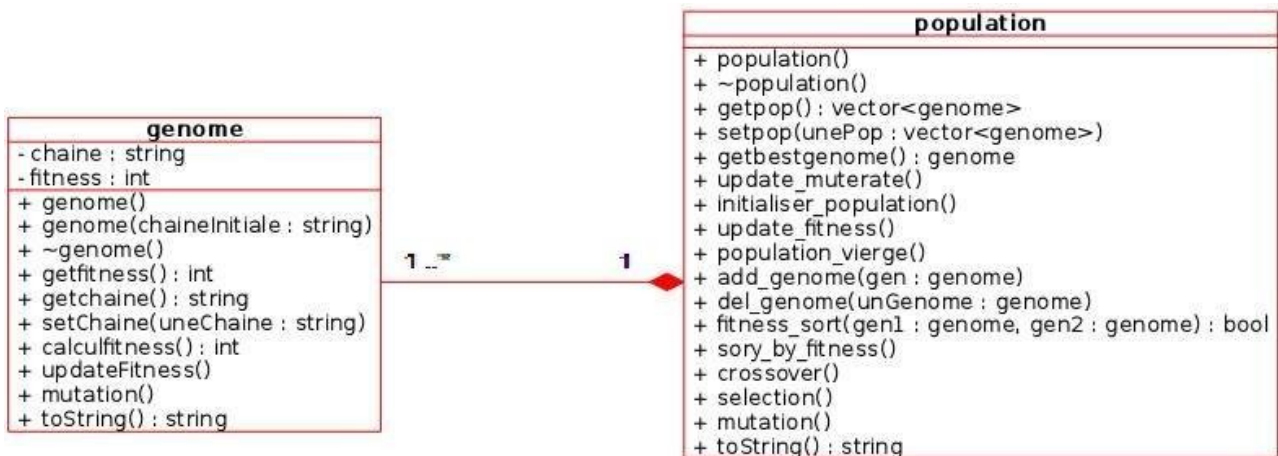


La désignation des cas d'utilisation parlent d'elle même :

- L'utilisateur doit rentrer les différents paramètres de l'application.
- Il peut ensuite lancer l'algorithme.

Description du domaine

Nous présentons ci-après le diagramme de classes de notre application. Conformément à notre précédente analyse, on y trouve 2 classes – 2 modules – implémentant les fonctionnalités recherchées.



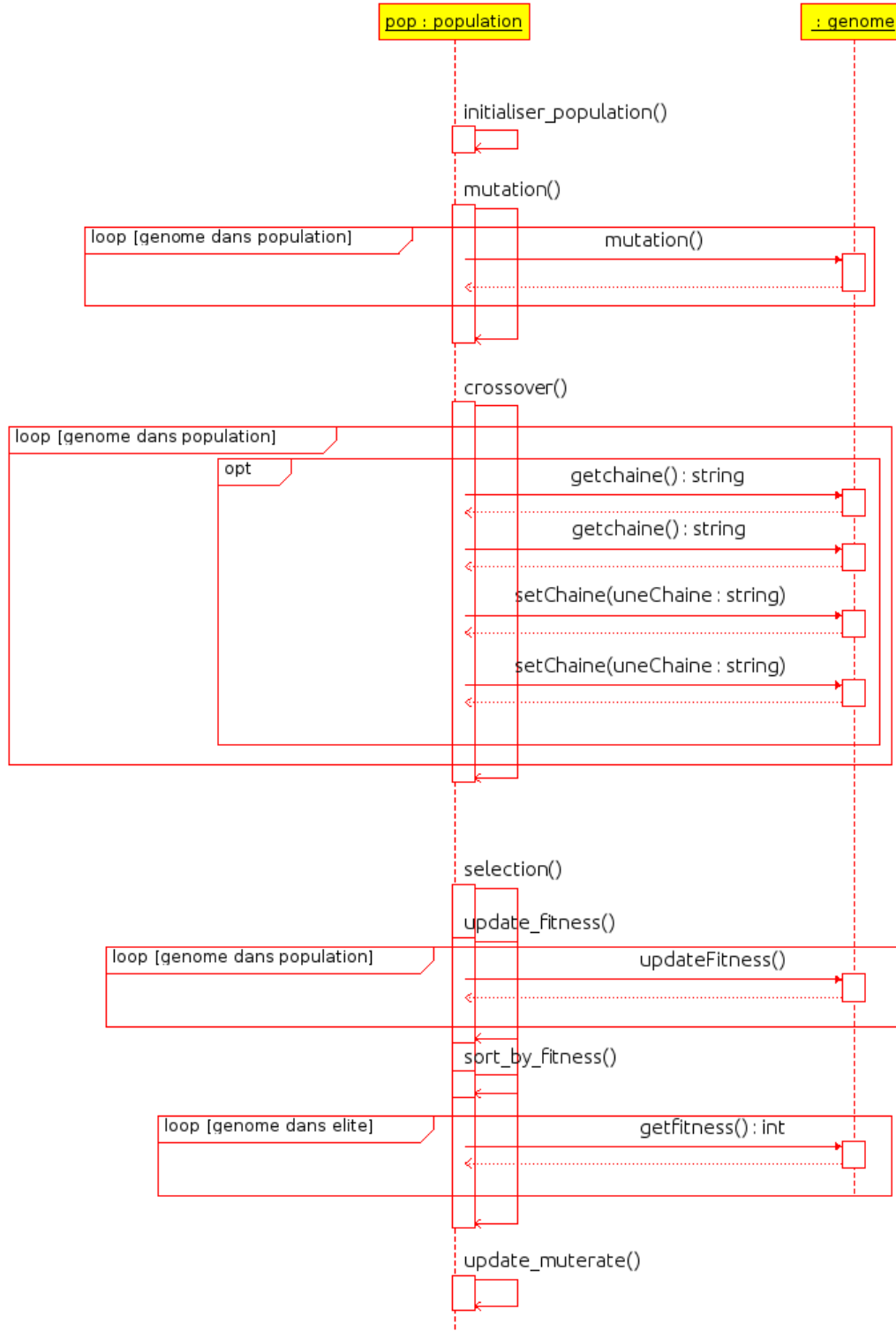
Interactions entre les composants (associations)

Nous avons ici qu'une seule interactions entre nos composants puisque nous n'en avons que deux.

La classe genome compose la classe population car bien évidemment, on veut que population représente un ensemble de génome, d'où aussi la multiplicité 1..*. On utilise ici la composition puisqu'il ne peut pas exister de population si il n'y a pas de génomes à exploiter.

Tests d'intégration

Nous avons choisis de présenter dans cette partie la communication entre les deux classes lors du déroulement de l'algorithme, objectif majeur du programme. L'entrée des paramètres étant élémentaire, nous ne trouvons pas cela intéressant de le représenter ici. Le diagramme de séquence se présente donc de la manière suivante :



Nous avons choisis de ne pas donner de nom à l'entité de *genome* puisque l'on interagit avec plusieurs objets de cette classe, ainsi il serait incorrect de mettre un seul nom.

Implémentation et test de l'algorithme

A la suite de ces analyses et modélisations, nous avons implémenté l'algorithme.

Nous avons donc 2 classes comme conceptualisées précédemment ainsi qu'un programme principal. Ce programme principal se charge de lancer l'algorithme génétique en suivant les étapes définies en présentation de celui-ci.

Comme nous l'avons dit, la première chose à faire est de trouver les paramètres qui nous donneront les meilleurs résultats. Plus une population contiendra de génomes, moins il lui faudra de générations pour arriver à une solution optimale, mais au coût du temps.

Ainsi, nous avons fait quelques recherches sur le sujet, qui nous ont amené aux conclusions suivantes :

- MUTERATE optimal : 0.001 et CROSSOVERRATE optimal : 0.6 et une taille de population entre 50 et 100 – DeJong (1975)
- Un MUTERATE supérieur à 0.05 est généralement néfaste à l'algorithme – Grefenstette (1986).
- Forgarty(1989) a aussi montré qu'un MUTERATE variable améliorerait considérablement les performances de l'algorithme.

Sachant tout ceci, voici les paramètres utilisés pour une phrase simple :

```
#ifndef GA_PARAMETERS_H
#define GA_PARAMETERS_H

#define target string("Hello world !")

#define SIZE target.size()
#define RANDOM (float)(rand()%100+1)/100
#define NBGENE 10000

//Paramètres du GA

#define ELITRATE 0.20
#define CROSSOVERRATE 0.6
#define NBGENOME 50

#define MUTERATEMAX 0.05
#define COEFFMUTERATE 1.005
#define DELTAFITNESS 1

static double MUTERATE=0.01;
```

On converge assez vite en nombre de générations, essayons de voir avec moins de génomes désormais :

```
#ifndef GA_PARAMETERS_H
#define GA_PARAMETERS_H

#define target string("Hello world !")

#define SIZE target.size()
#define RANDOM (float)(rand()%100+1)/100
#define NBGENE 10000

//Paramètres du GA

#define ELITRATE 0.20
#define CROSSEVERRATE 0.6
#define NBGENOME 10

#define MUTERATEMAX 0.05
#define COEFFMUTERATE 1.005
#define DELTAFITNESS 1

static double MUTERATE=0.01;

#endif
```

```
GASTring
2214 : chaine : Hello wo!ld ! | fitness : (12) Muterate actuel : 0,050077
2215 : chaine : Hello wo!ld ! | fitness : (12) Muterate actuel : 0,050077
2216 : chaine : Hello wo!ld ! | fitness : (12) Muterate actuel : 0,050077
2217 : chaine : Hello wo!ld ! | fitness : (12) Muterate actuel : 0,050077
2218 : chaine : Hello wo!ld ! | fitness : (12) Muterate actuel : 0,050077
2219 : chaine : Hello wo!ld ! | fitness : (12) Muterate actuel : 0,050077
2220 : chaine : Hello wo!ld ! | fitness : (12) Muterate actuel : 0,050077
2221 : chaine : Hello wo!ld ! | fitness : (12) Muterate actuel : 0,050077
2222 : chaine : Hello wo!ld ! | fitness : (12) Muterate actuel : 0,050077
2223 : chaine : Hello wo!ld ! | fitness : (12) Muterate actuel : 0,050077
2224 : chaine : Hello wo!ld ! | fitness : (12) Muterate actuel : 0,050077
2225 : chaine : Hello wo!ld ! | fitness : (12) Muterate actuel : 0,050077
2226 : chaine : Hello wo!ld ! | fitness : (12) Muterate actuel : 0,050077
2227 : chaine : Hello wo!ld ! | fitness : (12) Muterate actuel : 0,050077
2228 : chaine : Hello wo!ld ! | fitness : (12) Muterate actuel : 0,050077
2229 : chaine : Hello wo!ld ! | fitness : (12) Muterate actuel : 0,050077
2230 : chaine : Hello wo!ld ! | fitness : (12) Muterate actuel : 0,050077
2231 : chaine : Hello wo!ld ! | fitness : (12) Muterate actuel : 0,050077
2232 : chaine : Hello world ! | fitness : (13) Muterate actuel : 0,050077
Génération finale : 2232

Process returned 0 (0x0)   execution time : 0,355 s
Press ENTER to continue.
```

La convergence est alors bien meilleure en terme de temps d'exécution ici. Moins de génomes signifierait donc un temps d'exécution plus rapide ? Testons ça avec une phrase plus longue.

```
#ifndef GA_PARAMETERS_H
#define GA_PARAMETERS_H

#define target string("Hello world ! Ici Maxime, Killian et Rafael.")

#define SIZE target.size()
#define RANDOM (float)(rand()%100+1)/100
#define NBGENE 30000

//Paramètres du GA

#define ELITRATE 0.20
#define CROSSEVERRATE 0.6
#define NBGENOME 10

#define MUTERATEMAX 0.05
#define COEFFMUTERATE 1.005
#define DELTAFITNESS 1

static double MUTERATE=0.01;

#endif
```

```
GASTring
uterate actuel : 0,050077
29993 : chaine : Hel`o world ! Ich2Maxime, K,3!s\\n et yafael. | fitness : (36) M
uterate actuel : 0,050077
29994 : chaine : Hel`o world ! Ich2Maxime, K,3!s\\n et yafael. | fitness : (36) M
uterate actuel : 0,050077
29995 : chaine : Hel`o world ! Ich2Maxime, K,3!s\\n et yafael. | fitness : (36) M
uterate actuel : 0,050077
29996 : chaine : Hel`o world ! Ich2Maxime, K,3!s\\n et yafael. | fitness : (36) M
uterate actuel : 0,050077
29997 : chaine : Hel`o world ! Ich2Maxime, K,3!s\\n et yafael. | fitness : (36) M
uterate actuel : 0,050077
29998 : chaine : Hel`o world ! Ich2Maxime, K,3!s\\n et yafael. | fitness : (36) M
uterate actuel : 0,050077
29999 : chaine : Hel`o world ! Ich2Maxime, K,3!s\\n et yafael. | fitness : (36) M
uterate actuel : 0,050077
Génération finale : 30000

Process returned 0 (0x0)   execution time : 7,639 s
Press ENTER to continue.
```

Aie... Ici on atteint la limite de générations fixée, sans avoir trouvé de solution. Le nombre de génomes semble trop faible. Augmentons le pour voir :

```
#ifndef GA_PARAMETERS_H
#define GA_PARAMETERS_H

#define target string("Hello world ! Ici Maxime, Killian et Rafael.")

#define SIZE target.size()
#define RANDOM (float)(rand()%100+1)/100
#define NBGENE 30000

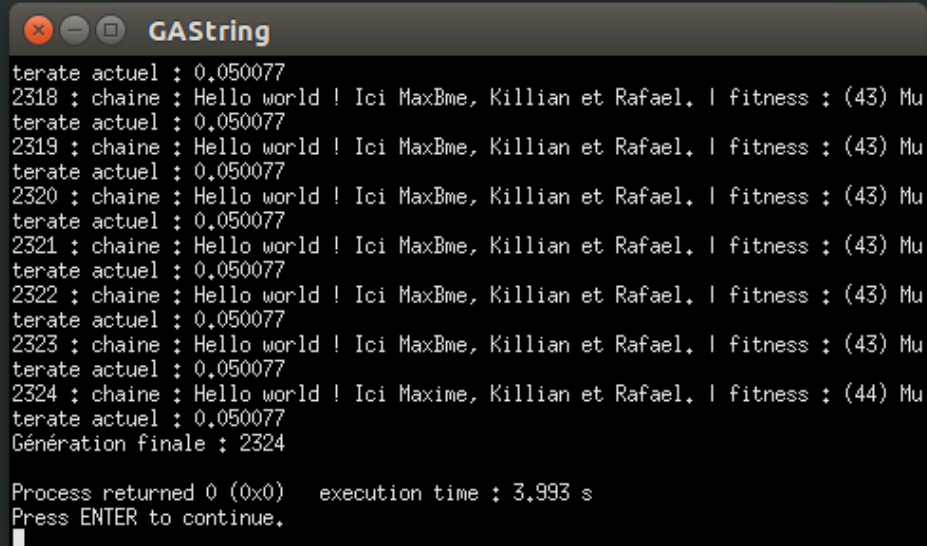
//Paramètres du GA

#define ELITRATE 0.20
#define CROSSOVERRATE 0.6
#define NBGENOME 40

#define MUTERATEMAX 0.05
#define COEFFMUTERATE 1.005
#define DELTAFITNESS 1

static double MUTERATE=0.01;

#endif
```



```
GAString
terate actuel : 0.050077
2318 : chaine : Hello world ! Ici MaxBme, Killian et Rafael. | fitness : (43) Mu
terate actuel : 0.050077
2319 : chaine : Hello world ! Ici MaxBme, Killian et Rafael. | fitness : (43) Mu
terate actuel : 0.050077
2320 : chaine : Hello world ! Ici MaxBme, Killian et Rafael. | fitness : (43) Mu
terate actuel : 0.050077
2321 : chaine : Hello world ! Ici MaxBme, Killian et Rafael. | fitness : (43) Mu
terate actuel : 0.050077
2322 : chaine : Hello world ! Ici MaxBme, Killian et Rafael. | fitness : (43) Mu
terate actuel : 0.050077
2323 : chaine : Hello world ! Ici MaxBme, Killian et Rafael. | fitness : (43) Mu
terate actuel : 0.050077
2324 : chaine : Hello world ! Ici Maxime, Killian et Rafael. | fitness : (44) Mu
Génération finale : 2324

Process returned 0 (0x0)   execution time : 3.993 s
Press ENTER to continue.
```

Cette fois, on trouve une solution en temps plus qu'acceptable. Le problème illustré avec cette situation est un de ceux énoncés précédemment : avec trop peu de génomes, on met beaucoup trop de temps à trouver les dernières lettres manquantes. En revanche, voilà ce qu'il se passe avec trop de génomes :

```
#ifndef GA_PARAMETERS_H
#define GA_PARAMETERS_H

#define target string("Hello world ! Ici Maxime, Killian et Rafael.")

#define SIZE target.size()
#define RANDOM (float)(rand()%100+1)/100
#define NBGENE 30000

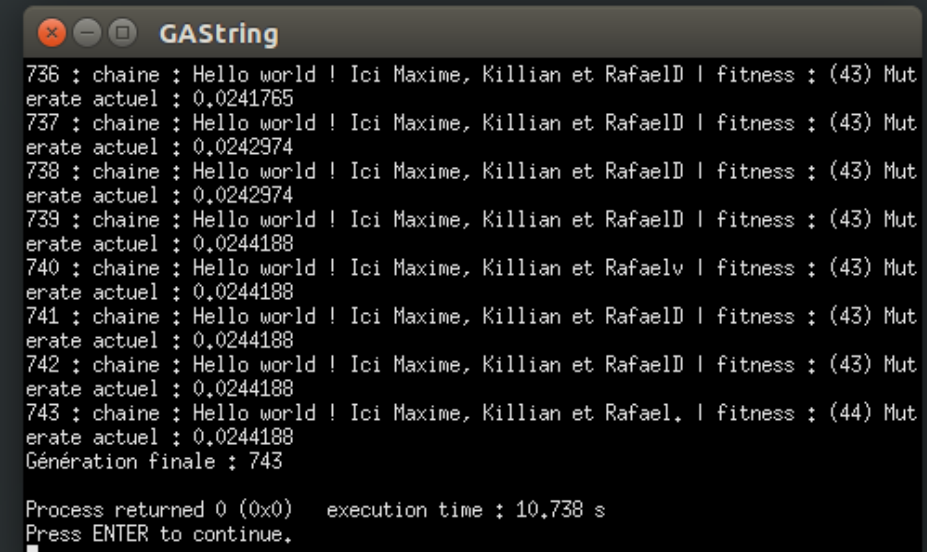
//Paramètres du GA

#define ELITRATE 0.20
#define CROSSOVERRATE 0.6
#define NBGENOME 100

#define MUTERATEMAX 0.05
#define COEFFMUTERATE 1.005
#define DELTAFITNESS 1

static double MUTERATE=0.01;

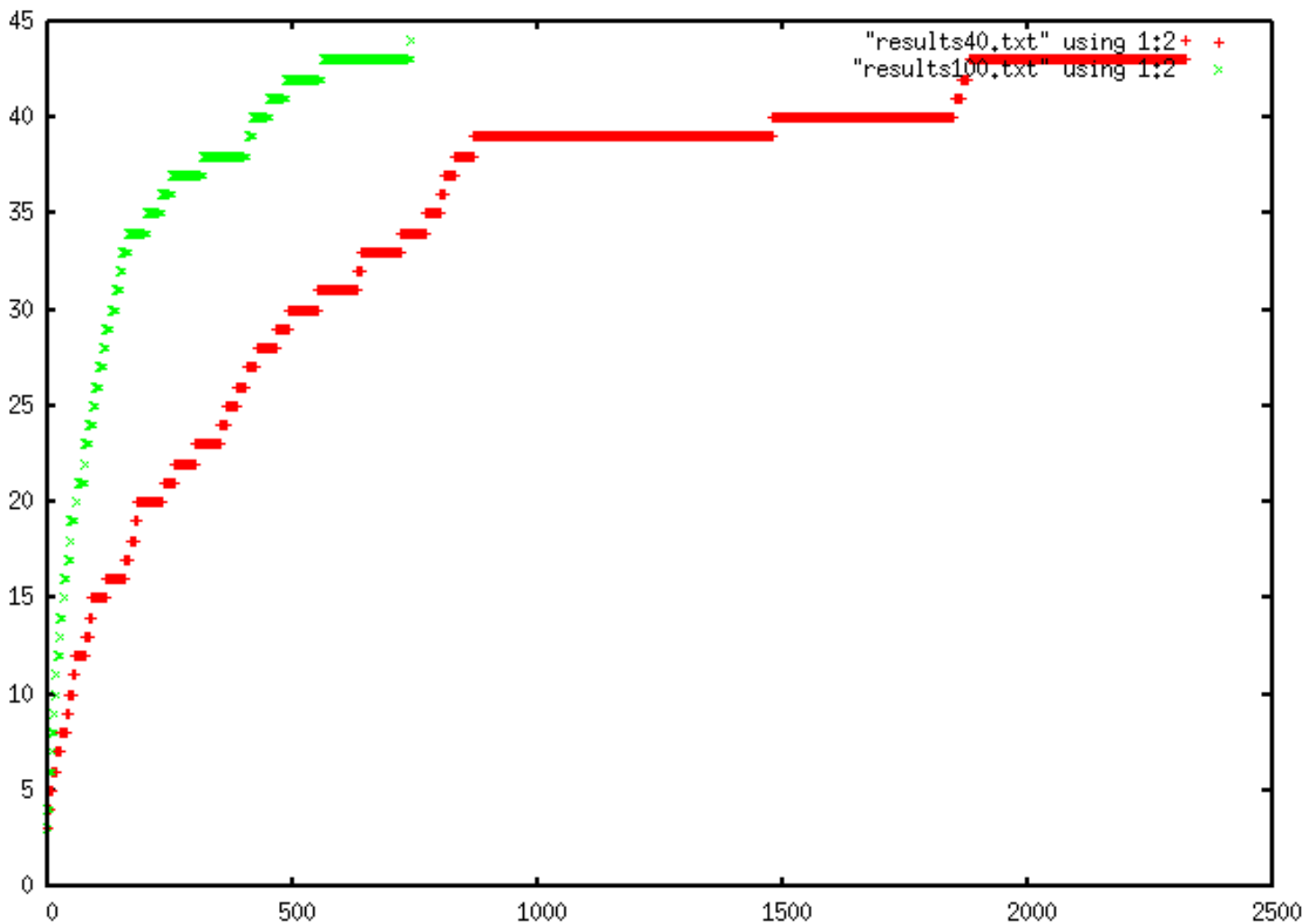
#endif
```



```
GAString
736 : chaine : Hello world ! Ici Maxime, Killian et RafaelD | fitness : (43) Mut
erate actuel : 0.0241765
737 : chaine : Hello world ! Ici Maxime, Killian et RafaelD | fitness : (43) Mut
erate actuel : 0.0242974
738 : chaine : Hello world ! Ici Maxime, Killian et RafaelD | fitness : (43) Mut
erate actuel : 0.0242974
739 : chaine : Hello world ! Ici Maxime, Killian et RafaelD | fitness : (43) Mut
erate actuel : 0.0244188
740 : chaine : Hello world ! Ici Maxime, Killian et Rafaelv | fitness : (43) Mut
erate actuel : 0.0244188
741 : chaine : Hello world ! Ici Maxime, Killian et RafaelD | fitness : (43) Mut
erate actuel : 0.0244188
742 : chaine : Hello world ! Ici Maxime, Killian et RafaelD | fitness : (43) Mut
erate actuel : 0.0244188
743 : chaine : Hello world ! Ici Maxime, Killian et Rafael. | fitness : (44) Mut
erate actuel : 0.0244188
Génération finale : 743

Process returned 0 (0x0)   execution time : 10.738 s
Press ENTER to continue.
```

Afin de comparer les 2 vitesses de convergence, voici un graphe représentant l'évolution de la meilleure fitness au cours du temps :



Ce graphe nous montre bien comment un plus grand nombre de génomes empêche l'algorithme de stagner trop près de la solution finale, comme le fait la courbe rouge. Cependant, cela se fait au prix du temps de calcul, qui lui explose dans le cas des 100 génomes.

Ce graphe montre aussi le caractère de convergence logarithmique de notre algorithme, ce qui était à prévoir : il est facile de trouver des lettres au début, mais plus on s'approche de la fin, plus cela devient compliqué, même avec un grand nombre de génomes. A noter qu'au delà de 200 génomes, le calcul est extrêmement lent, au point de ne pas être utilisable. On note donc la corrélation entre le nombre de génomes et la longueur de la phrase. Ainsi, mettons une formule plus générique pour déterminer le nombre de génomes, et essayons désormais une phrase beaucoup plus longue :

```

#ifndef GA_PARAMETERS_H
#define GA_PARAMETERS_H

#define target string("Hello world ! Ici Maxime Aberton, Killian Jaubert et Rafael Cartenet. Quel temps fait-il ?")

#define SIZE target.size()
#define RANDOM (float)(rand()%100+1)/100
#define NBGENE 30000

//Paramètres du GA

#define ELITRATE 0.20
#define CROSSOVERRATE 0.6
#define NBGENOME (SIZE%2==1?SIZE+1:SIZE)

#define MUTERATEMAX 0.04
#define COEFFMUTERATE 1.008
#define DELTAFITNESS 1

static double MUTERATE=0.01;

#endif

```

GAString

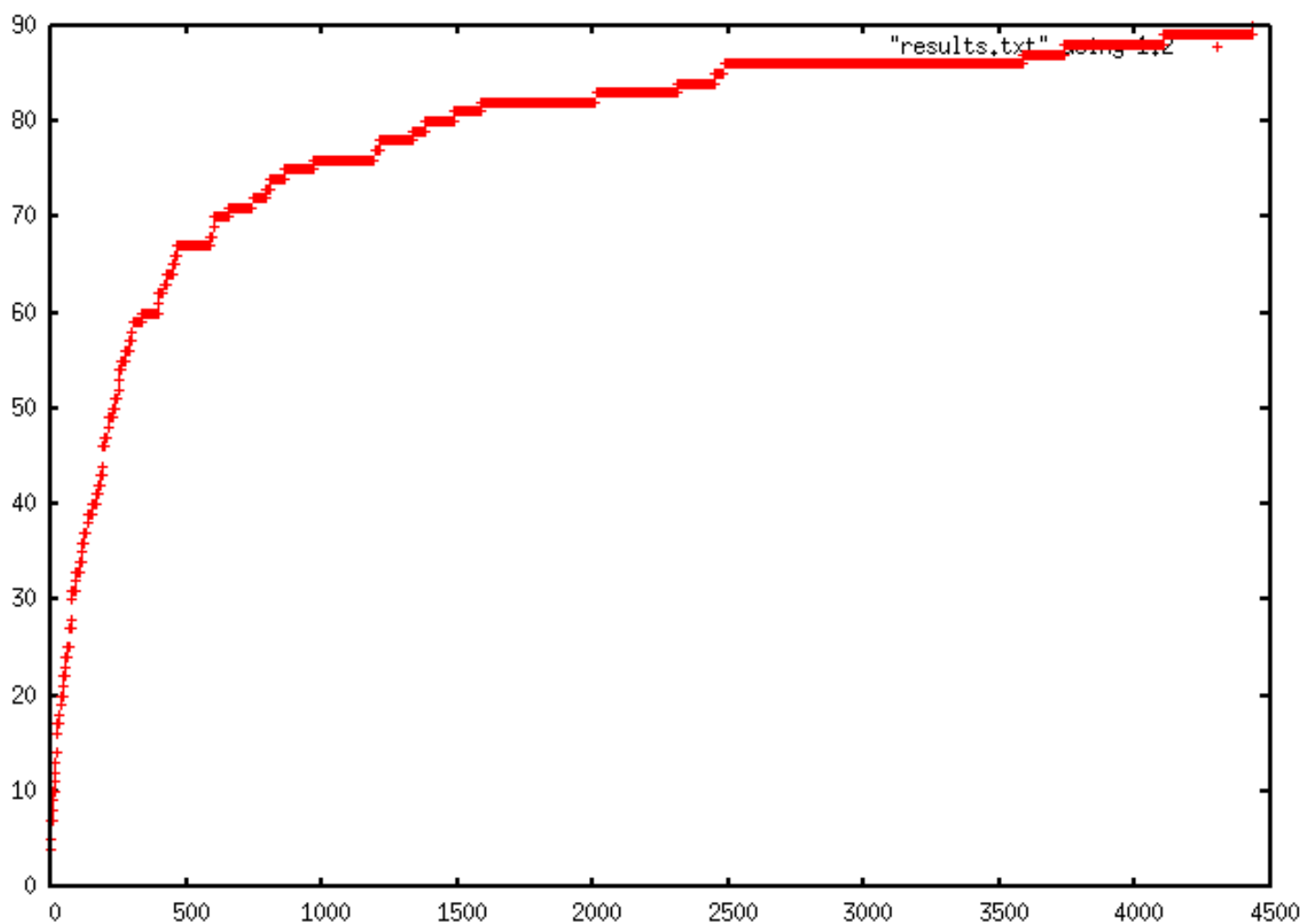
```

4427 : chaine : Hello world ! Ici Maxime Aberton, Killian Jaubert et Rafael Cartenet. Quel temps
fait-il ? | fitness : (89/90) Muterate actuel : 0,0346612
4428 : chaine : Hello world ! Ici Maxime Aberton, Killian Jaubert et Rafael Cartenet. Quel temps
fait-il ? | fitness : (89/90) Muterate actuel : 0,0346612
4429 : chaine : Hello world ! Ici Maxime Aberton, Killian Jaubert et Rafael Cartenet. Quel temps
fait-il ? | fitness : (89/90) Muterate actuel : 0,0346612
4430 : chaine : Hello world ! Ici Maxime Aberton, Killian Jaubert et Rafael Cartenet. Quel temps
fait-il ? | fitness : (89/90) Muterate actuel : 0,0346612
4431 : chaine : Hello world ! Ici Maxime Aberton, Killian Jaubert et Rafael Cartenet. Quel temps
fait-il ? | fitness : (89/90) Muterate actuel : 0,0346612
4432 : chaine : Hello world ! Ici Maxime Aberton, Killian Jaubert et Rafael Cartenet. Quel temps
fait-il ? | fitness : (89/90) Muterate actuel : 0,0346612
4433 : chaine : Hello world ! Ici Maxime Aberton, Killian Jaubert et Rafael Cartenet. Quel temps
fait-il ? | fitness : (89/90) Muterate actuel : 0,0346612
4434 : chaine : Hello world ! Ici Maxime Aberton, Killian Jaubert et Rafael Cartenet. Quel temps
fait-il ? | fitness : (90/90) Muterate actuel : 0,0346612
Génération finale : 4434

Process returned 0 (0x0)   execution time : 57,060 s
Press ENTER to continue.

```

On arrive à retrouver une phrase d'un peu moins de 100 caractères en moins d'une minute ! Voici la courbe de progression des fitness :



Répartition des tâches

Au début du projet, nous souhaitions réaliser un réseau de neurones sur un sujet simple, mais le concept de réseau neuronal est extrêmement complexe et ne se prêtait pas vraiment à un mini-projet de C++. Nous avons donc opté pour un algorithme génétique, ce qui est plus simple à implémenter, tout aussi intéressant et qui nous permettrait de progresser en C++.

Nous avons donc tous ensemble effectué des recherches sur le sujet, et sommes tombés d'accord pour coder cet algorithme de recherche de chaîne de caractères.

Nous avons analysé ensemble les besoins de notre projet. Il en est de même pour la conception étant donné que nous n'avions pas désigné de « chef » pour s'en occuper.

Nous nous sommes repartis les différentes méthodes à coder étant donné que le projet est assez modulable. A la fin, nous avons mis en commun notre travail et corrigé les erreurs éventuelles. Nous avons ensuite réfléchi ensemble à l'optimisation de l'algorithme. C'est à ce moment que nous avons effectué différentes recherches pour trouver les paramètres optimaux et effectuer les tests de l'algorithme.

Conclusion

Le principe de l'algorithme est donc relativement simple et nous l'avons utilisé pour un cas de figure qui l'est encore plus – la recherche de chaîne de caractères. Mais beaucoup de fonctions sont réutilisables si nous souhaitons réutiliser un algorithme génétique pour un autre projet. Il ne faudrait alors changer que la modélisation du problème et éventuellement adapter quelques unes des fonctions.

En ce qui concerne le projet en lui-même, on constate que la qualité des paramètres est ce qui rend l'algorithme plus ou moins performant. Nous nous sommes servis de recherches qui montraient que le set de paramètres que l'on a utilisé est performant, ce qui se confirme par les bons résultats observés. Si l'on voulait optimiser les paramètres au maximum, on pourrait faire un autre algorithme génétique qui calculerait cette fois les paramètres de l'algorithme principal.

Nous nous sommes intéressés avec ce projet à un domaine de l'informatique que nous connaissions peu et que nous désirions connaître plus en détails. Nous avons aussi développer nos compétences en C++ en appliquant les enseignements du cours.