

# DE LA BIOLOGIE DANS DU CODE : LES ALGORITHMES GÉNÉTIQUES

par Tristan Colombo

**Existe-t-il un lien entre un être vivant, aussi simple soit-il, et un programme ?**  
**On serait tenté de répondre immédiatement non... Mais en fait, la réponse ne peut pas être aussi catégorique, tout dépend de la façon dont a été codé le programme !**

**D**e tout temps, l'Homme s'est inspiré de la nature pour ses inventions : la fermeture Velcro est née de l'observation des fruits secs de la bardane, qui comportent de minuscules crochets très recourbés et qui s'accrochent à n'importe quel tissu ; le gilet pare-balles emprunte sa structure à la soie des fils d'araignée, qui sont dix fois plus résistants que l'acier (et un peu plus légers aussi), etc. C'est de la bionique : l'utilisation de modèles vivants en vue de réalisations techniques (comme pour Steve Austin, l'homme qui valait trois milliards). Pourquoi donc ne pas puiser dans la nature l'inspiration pour créer des méthodes de résolution de problèmes ? C'est ce qui a été fait avec notamment les algorithmes génétiques, qui sont une sous-classe des algorithmes évolutionnaires [1], également appelés algorithmes évolutionnistes ou AE pour les intimes. Les algorithmes génétiques (GA) sont apparus en 1975 [2] et ils sont les

précurseurs des algorithmes évolutionnistes, bien qu'en tant que sous-classe on puisse penser le contraire : la généralisation est intervenue postérieurement.

À quoi servent ces algorithmes ? Ils sont issus d'un domaine très particulier de l'informatique théorique : l'optimisation calculatoire, qui consiste à trouver les meilleures solutions pour un problème donné. Tout le monde a déjà entendu parler du problème du voyageur de commerce (*Travelling Salesman Problem* ou TSP en anglais), exposé pour la première fois par W. Hamilton en 1859. La résolution de ce problème où un marchand doit parcourir toutes les villes d'un pays en effectuant le plus court chemin peut trouver une solution grâce à l'optimisation. Ainsi, les algorithmes génétiques permettent d'identifier au moins une solution à un problème d'optimisation, sans garantir que la solution soit optimale (c'est la définition de ce que l'on nomme des heuristiques). En fait, on peut même

aller plus loin en disant que les algorithmes génétiques constituent une stratégie générale qui peut être appliquée à un grand nombre de problèmes (à condition de les représenter correctement). On parle alors de mét-heuristiques. Attention toutefois, car les algorithmes génétiques sont lents et leur usage ne se justifie que si l'application d'algorithmes plus efficaces n'est pas possible.

Nous avons défini le côté « algorithme » des algorithmes génétiques, mais qu'en est-il de la « génétique » ? On retrouve ici l'inspiration de la nature, mais d'un point de vue microscopique. La théorie de l'évolution, introduite par Charles Darwin en 1859 [3], nous a apporté la sélection naturelle : un être vivant adapté à son milieu à plus de chance de survivre et donc, de se reproduire et de propager ses caractéristiques. Si l'on ajoute à cela la théorie de l'hérédité de Gregor Mendel [4] et la génétique des populations [5], on aboutit à la théorie synthétique de

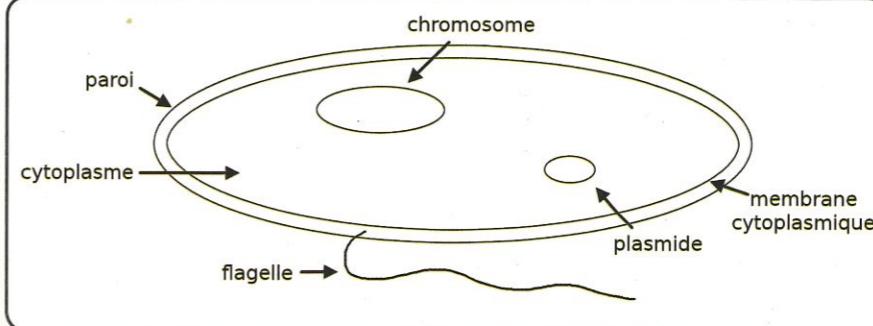


Fig. 1 : Schéma simplifié de la structure bactérienne

l'évolution [6], base des algorithmes génétiques. Nous aurons donc besoin d'un minimum de connaissances en biologie et plus précisément en génétique pour comprendre ces algorithmes. Nous commencerons par une petite parenthèse expliquant les mécanismes de l'évolution avant d'aborder les algorithmes génétiques et de les mettre en pratique sur un exemple simple.

## 1 | Un petit peu de biologie

Dans cette partie, nous aborderons les principes biologiques nécessaires à la compréhension des GA. Il y a deux façons d'envisager cet exposé : soit on se limite aux définitions des éléments utilisés, soit on s'intéresse un peu plus à la biologie en essayant de vraiment comprendre ce qui se passe. J'ai choisi la deuxième solution. Comme nous sommes sur un domaine très restreint, je tiens à vous rassurer, ça ne sera vraiment pas compliqué... Mais différent de ce que vous avez l'habitude de lire.

### 1.1 Les bactéries

Les algorithmes génétiques s'appuient sur un modèle particulier d'êtres vivants : les bactéries. De l'ordre du micromètre, les bactéries sont les plus petits organismes vivants (les virus, qui sont plus petits que les bactéries, ne peuvent pas être considérés comme vivants de par leur incapacité à se reproduire seuls). Une bactérie est une cellule entourée d'une membrane et contenant tous les éléments nécessaires à sa propre reproduction. Les populations de bactéries peuvent s'adapter aux variations de leur environnement, ou adopter en quelques générations seulement de nouveaux caractères.

D'un point de vue physique, l'architecture d'une bactérie est relativement simple comme le montre la figure 1. La cellule est tout d'abord entourée d'une paroi qui lui donne sa forme, sa résistance, et qui entoure une seconde enveloppe plus ou moins épaisse, la membrane cytoplasmique. Le (ou les) chromosome(s) et éventuellement le (ou les) plasmide(s) qui sont des sortes de petits chromosomes circulaires, porteurs de l'information génétique, se trouvent dans l'environnement délimité par la membrane cytoplasmique : le cytoplasme. Dans le cadre des GA, nous allons simplifier ce modèle en considérant que l'information génétique n'est portée que par un unique **chromosome**.

Ce chromosome est composé de molécules d'ADN, dont la fameuse structure en « double hélice » fut découverte en 1953 [7]. Elle est formée par une succession de petites unités appelées **nucléotides**, constituées de trois éléments : une molécule d'acide phosphorique, un sucre et une base organique. Dans le cas de l'ADN, le sucre est du désoxyribose, d'où son nom : Acide(phosphorique) Désoxyribo(se) Nucléique. Les **bases**, quant à elles, sont au nombre de quatre : l'adénine (A), la thymine (T), la cytosine (C) et la guanine (G). Ces chaînes ont trois propriétés essentielles :

- Elles sont hélicoïdales : elles s'enroulent autour d'un axe en formant une double hélice droite ;
- Elles sont antiparallèles : les deux brins de nucléotides sont parallèles, mais se lisent dans des directions opposées ;
- Elles sont complémentaires : les bases des deux brins ne pouvant

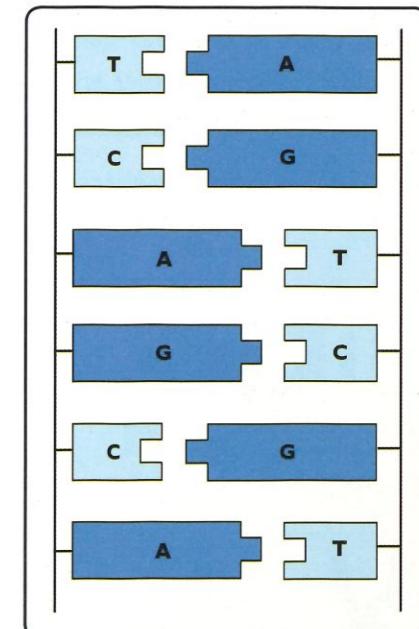


Fig. 2 : Les deux chaînes de nucléotides d'une molécule d'ADN. Les paires de bases constituant chaque barreau ont la même taille.

pas s'apparier n'importe comment, un peu comme pour les pièces d'un jeu de Lego, les deux bases C et T forment de petites briques, alors que les bases A et G forment de grandes briques. On ne peut associer qu'une petite brique avec une grande brique. Les seuls couples autorisés sont A-T et C-G (voir figure 2 page précédente).

L'ensemble des informations génétiques est contenu dans ce que l'on appelle le **génome**. La séquence **ACCCGAT** est un exemple d'information qui pourra ensuite être traduite sous forme de protéine pour constituer un être vivant et déterminer ainsi son apparence ou **phénotype**.

## 1.2 Évolution

Intéressons-nous maintenant à l'évolution de cette information. La reproduction des bactéries prend la forme d'un procédé asexué où chacune grandit en taille, duplique son chromosome, puis se divise en deux bactéries identiques ou clones. Malgré ce type de reproduction clonal, qui laisserait supposer qu'une bactérie fille soit identique à sa « mère », l'information génétique est modifiée d'une génération à l'autre. On peut par exemple s'apercevoir qu'une bactérie pathogène devient résistante aux antibiotiques qui la combattent. Mais quel est ce miracle ? Plusieurs cas pouvant apparaître, nous n'aborderons ici que ceux qui seront utilisés dans les algorithmes génétiques.

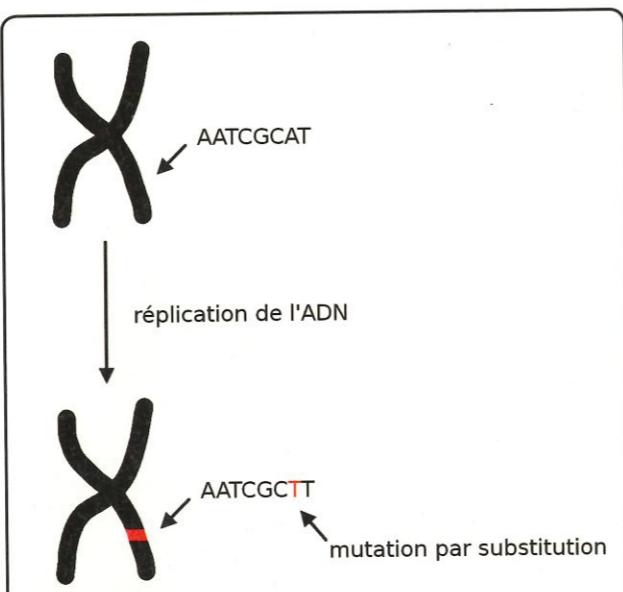


Fig. 4 : Recombinaison génétique par mutation (substitution)

### 1.2.1 Crossing-over

Le **crossing-over**, ou entrecroisement en bon français, a lieu lors de la duplication du chromosome : deux chromosomes parents se touchent en un point et échangent leurs informations à partir de ce point, comme le montre la figure 3.

### 1.2.2 Mutations

Lors de la réPLICATION de l'ADN, il peut y avoir des ratés à cause d'accidents de copie des bases. On parle alors de mutations et on en distingue trois types :

- la **substitution** : une base est mal copiée ;
- la **délétion** : une base est oubliée ;
- l'**insertion** : une base est ajoutée.

Ces accidents de copie sont tout à fait analogues à une faute de frappe qui se produirait lors de la copie d'un texte. Prenons par exemple deux phrases : d'un côté « La reine s'en est allée » et de l'autre « La peine s'en est allée ». Ces deux phrases, bien que sémantiquement et syntaxiquement correctes, n'ont pas le même sens. Une seule lettre a été modifiée entre les deux et dans un cas le chagrin s'est évaporé, alors que dans l'autre la femme du roi est partie. Au niveau de l'ADN, les mutations peuvent avoir une influence sur la protéine qui sera produite et engendrer un phénotype différent et donc, un comportement différent en fonction de

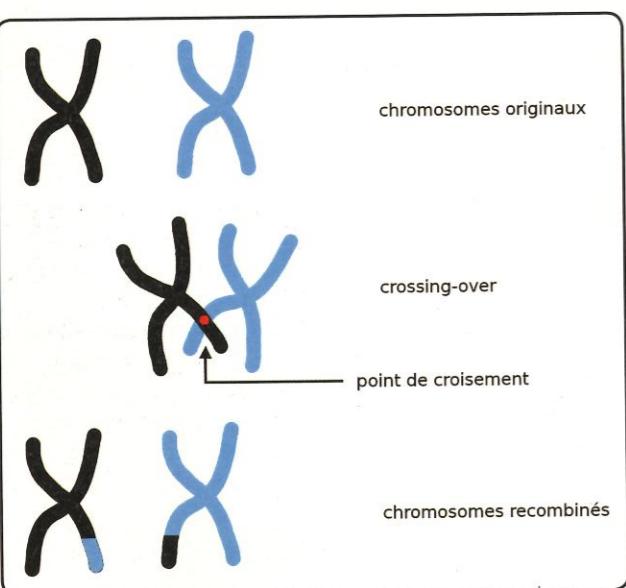


Fig. 3 : Recombinaison génétique par crossing-over

# ABONNEZ-VOUS !

CONSULTEZ L'ENSEMBLE DE NOS OFFRES SUR : [boutique.ed-diamond.com](http://boutique.ed-diamond.com) !

11 Numéros de  
**GNU/Linux Magazine**



**60€\***

au lieu de **86,90 €\***  
en kiosque  
Économie  
**26,90€**  
Économisez **plus de 30%\***

\* Sur le prix de vente unitaire France Métropolitaine

OFFRE VALABLE UNIQUEMENT EN FRANCE MÉTROPOLITaine. Pour les tarifs hors France Métropolitaine, consultez notre site : [boutique.ed-diamond.com](http://boutique.ed-diamond.com)

**NOUVEAU !** Abonnez-vous (réabonnez-vous) en ligne sur :  
**boutique.ed-diamond.com**

Vous pouvez ainsi : ➔ Avoir accès à votre suivi personnalisé d'abonnement ➔ Profiter des promos réservées à nos abonnés ➔ Vous réabonner facilement sans interruption d'abonnement

Pour plus d'informations, veuillez nous contacter via e-mail : [cial@ed-diamond.com](mailto:cial@ed-diamond.com) ou par téléphone : +33 (0)3 67 10 00 20

Bon d'abonnement à découper et à renvoyer à l'adresse ci-dessous

Tournez SVP pour découvrir toutes les offres d'abonnement >>

Voici mes coordonnées postales :

Société :	
Nom :	
Prénom :	
Adresse :	
Code Postal :	
Ville :	
Pays :	
Téléphone :	
E-mail :	

- Je souhaite recevoir les offres promotionnelles et newsletters des Éditions Diamond.  
 Je souhaite recevoir les offres promotionnelles des partenaires des Éditions Diamond.

En envoyant ce bon de commande, je reconnais avoir pris connaissance des conditions générales de vente des Éditions Diamond à l'adresse internet suivante : [boutique.ed-diamond.com/content/3-conditions-generales-de-ventes](http://boutique.ed-diamond.com/content/3-conditions-generales-de-ventes) et reconnais que ces conditions de vente me sont opposables.



Édité par Les Éditions Diamond  
Service des Abonnements  
B.P. 20142 - 67603 Sélestat Cedex  
Tél. : + 33 (0) 3 67 10 00 20  
Fax : + 33 (0) 3 67 10 00 21

Vos remarques :

Tournez SVP pour découvrir toutes les offres d'abonnement >>



# ABONNEMENT GLMF

→ Tous les abonnements incluant GNU/Linux Magazine :



**NOUVEAU !** Abonnez-vous (réabonnez-vous) en ligne sur : [boutique.ed-diamond.com](http://boutique.ed-diamond.com)

Vous pouvez ainsi :  Avoir accès à votre suivi personnalisé d'abonnement

Profiter des promos réservées à nos abonnés

Vous réabonner facilement sans interruption d'abonnement

Pour plus d'informations, veuillez nous contacter via e-mail : [cial@ed-diamond.com](mailto:cial@ed-diamond.com) ou par téléphone : +33 (0)3 67 10 00 20

## → Nos Tarifs

s'entendent TTC et en euros

	F	OM1	OM2	E	RM
LM	Abonnement GLMF	France Métro.	Outre-Mer	Europe	Reste du Monde
		60 €	75 €	96 €	83 €
LM+	Abonnement GLMF + GLMF HS (6 Guides)			160 €	173 €
T	Abonnement GLMF + MISC + OS + LP + LE	198 €	253 €	325 €	276 €
T+	Abonnement GLMF + GLMF HS (6 Guides) + MISC + MISC HS + OS + LP + LP HS (3 Guides) + LE	299 €	382 €	491 €	415 €
					448 €

\* OM1 : Guadeloupe, Guyane française, Martinique, Réunion, St-Pierre-et-Miquelon, Mayotte

\* OM2 : Nouvelle Calédonie, Polynésie française, Wallis et Futuna, Terres Australes et Antarctiques françaises

## MA FORMULE D'ABONNEMENT :

Offre	Zone	Tarif
<input checked="" type="checkbox"/> LM		
<input checked="" type="checkbox"/> LM+		
<input checked="" type="checkbox"/> T		
<input checked="" type="checkbox"/> T+		

J'indique la somme due : (Total)

Exemple :  
Je souhaite m'abonner à l'ensemble des magazines + tous les Hors-séries/Guides et je vis en Belgique.  
Je coche donc l'offre T+ (la totale avec tous les Hors-Séries/Guides), puis ma zone (E), le montant sera donc de 415 euros.

### Je choisis de régler par :

- Chèque bancaire ou postal à l'ordre des Éditions Diamond (uniquement France et DOM TOM)
- Pour les règlements par virements, veuillez nous contacter via e-mail : [cial@ed-diamond.com](mailto:cial@ed-diamond.com) ou par téléphone : +33 (0)3 67 10 00 20

Date et signature obligatoires



[www.gnulinuxmag.com](http://www.gnulinuxmag.com)

ADN

l'environnement. La sélection naturelle fera ensuite le reste pour déterminer si cette mutation a été profitable à l'organisme ou non. La figure 4 montre une mutation par substitution sur un chromosome.

Voilà, ce sont les seules connaissances biologiques dont nous aurons besoin. Nous pouvons maintenant passer à la partie plus théorique.

## 2 Un petit peu de théorie

Commençons par le principe général :

- On crée une population au hasard : c'est la **population initiale**. Chaque individu possède un gène propre qui représente une solution possible au problème de départ. Par « solution possible », on entend plutôt proposition de solution, il ne s'agit pas d'une solution qui résout le problème (sinon il serait inutile de créer un algorithme...).
- Jusqu'à ce qu'un critère d'arrêt soit vérifié (nombre de générations ou temps d'exécution maximum par exemple) :

- On sélectionne une partie de la population en fonction d'une note qu'on lui attribue. Les nouveaux individus sont donc globalement mieux adaptés à leur environnement.
- Les individus sélectionnés se reproduisent et donnent naissance à une nouvelle génération qui remplace la population initiale. C'est ici qu'interviennent les recombinaisons génétiques amenant de la diversité dans les solutions.

Plus formellement, soit  $S$  l'espace de recherche des solutions et une fonction d'adaptation  $f$ , souvent appelée *fonction de fitness*, qui fournit une évaluation de la performance des individus et participe donc à la sélection et à la reproduction des individus,  $\text{score\_max}$  le meilleur score attribué par la fonction  $f$ ,  $\text{max\_it}$  le nombre maximal d'itérations, un algorithme génétique consiste à effectuer les étapes suivantes :

```

Construire la population initiale P0
i <- 0
Tant que f(P0) < score_max et i < max_it Faire
    Mi <- (f(Pi), Pi)
    Mi <- selection(Mi)
    Pi+1 <- crossingover(Mi)
    Pi+1 <- mutation(Pi+1)
    i <- i +1
  
```

Il est évident ici qu'au plus il y aura de générations, au plus nous aurons des individus (solutions) adaptés à leur environnement (problème de départ). Mais comme l'écrivait Montesquieu, « le mieux est le mortel ennemi du bien »... Il faudra parfois être raisonnable et arrêter l'algorithme avec une bonne solution qui ne sera pas forcément la meilleure.

Les problèmes pour implémenter cet algorithme seront les suivants :

- Comment représenter l'information génétique ?
- Comment déterminer la population initiale ?
- Comment faire intervenir le mécanisme de sélection pour obtenir les parents de la génération suivante ?
- Comment effectuer la reproduction des individus en y intégrant les recombinaisons génétiques ?

Nous allons voir chacun de ces points.

### 2.1 Représentation de l'information génétique

Même si ce n'est pas obligatoire, le génotype peut utiliser un codage binaire et il est donc représenté par une chaîne de bits. Il faudra alors disposer d'un fonction capable d'exprimer le phénotype d'un individu comme le montre le tableau suivant :

Génotype	Phénotype
00	**
01	*****
10	*****
11	*****

### 2.2 Calcul de la population initiale

Le choix de la population initiale est très important, car c'est de sa composition que dépendra la convergence plus ou moins rapide vers l'ensemble de solutions. Il faut choisir une population d'individus non homogène, qui soit répartie sur tout l'espace  $S$ . En utilisant un codage binaire, le plus simple consiste à générer de manière aléatoire uniforme cette population de départ. On assure ainsi une uniformité de répartition des génotypes... Mais pas forcément des phénotypes, si plusieurs gènes représentent la même donnée (10 et 01 pourraient avoir pour même phénotype une barre composée de quatre étoiles).

## 2.3 Sélection

La sélection s'effectue sur la base de la fonction de fitness appliquée à la population courante (notée **P<sub>i</sub>** dans l'algorithme). Il existe de nombreuses méthodes de sélection des individus, dont voici les principales :

- La sélection **par élitisme** : on ne conserve que les meilleurs individus au sens de la fonction de fitness ;
- La sélection **par roulette** [8] : on donne à chaque individu une probabilité d'être sélectionné proportionnelle à sa performance. En d'autres termes, on applique la fonction de fitness **f** à tous les individus, ce qui nous permet d'établir une hiérarchie ; puis, on calcule la probabilité de choisir un individu **I** en divisant **f(I)** par la somme des valeurs de **f** calculées sur la population courante. On sélectionne ensuite les individus à partir des probabilités calculées.
- La sélection **par rang** : méthode similaire à la sélection par roulette, mais la probabilité est calculée sur le rang de l'individu.
- La sélection **par tournoi** [9] : on crée arbitrairement des groupes de **n** individus (**n >= 2**) et on applique une autre méthode de sélection (par exemple l'une des trois précédentes) pour sélectionner les individus les plus performants.

## 2.4 Reproduction

La reproduction est assez simple. En fonction d'un pourcentage qu'il faudra fixer arbitrairement, on va appliquer des recombinaisons génétiques par crossing-over et mutation. Si l'on considère que la représentation des génomes se fait de manière binaire, un crossing-over s'obtiendra de la façon suivante :

<b>Parent 1</b>	1101 0011
<b>Parent 2</b>	0100 1010
<b>Enfant 1</b>	1101 1010
<b>Enfant 2</b>	0100 0011

La mutation, elle, se fera de manière encore plus rapide en sélectionnant une base de manière aléatoire :

<b>Parent</b>	1101 0011
<b>Enfant</b>	1111 0011

## 2.5 Paramétrage simple

Vous vous en êtes rendu compte : certains paramètres devront être fixés de manière totalement arbitraire. Malheureusement, il n'existe pas de paramétrage « universel »

qui donnerait de bons résultats avec n'importe quel problème. Voici des valeurs qui peuvent être utilisées comme point de départ avant de procéder à un éventuel réajustement :

<b>Population</b>	entre 50 et 100 individus
<b>Taux de crossing-over</b>	entre 70% et 95%
<b>Taux de mutations</b>	entre 0,001% et 0,1%

Nous avons globalement vu toute la théorie des algorithmes génétiques. Il nous reste à les appliquer à un exemple simple pour bien en comprendre le fonctionnement.

## 3 Un algorithme génétique simple

Dans cette partie, le code résolvant le problème sera donné en Python 3. Pourquoi encore ce langage ? La réponse tient dans le fait qu'il est très simple à comprendre et que, même ceux d'entre vous qui ne le connaissent pas pourront aisément adapter le code dans leur langue favori. J'ai choisi cette solution plutôt que de ne présenter qu'un algorithme, de manière à ce que vous puissiez vraiment tester directement le code et voir évoluer les résultats.

### 3.1 Le problème

Nous allons essayer de résoudre un problème simple : étant donnés les chiffres de 0 à 9 et les opérateurs numériques +, -, / et % (reste de la division entière), trouver une expression qui représentera une valeur cible. La multiplication n'est pas utilisée pour éviter d'obtenir des nombres trop grands, mais dans l'absolu, si vous le souhaitez, vous pouvez remplacer le reste de la division entière par la multiplication.

Cherchons par exemple des expressions qui valent 1234. Pour réduire l'espace de recherche, nous limiterons les expressions à 12 symboles maximum. Voici quelques solutions possibles :

- 1234
- 1200 + 30 + 4
- 2460 / 2 + 4
- ...

Le problème est posé, voyons comment le représenter.

## 3.2 Le codage

Nous avons vu que le codage binaire était ce qu'il y a de plus simple pour représenter le génotype. Dans le cas de notre problème, nous devons pouvoir représenter 14 symboles (les dix chiffres et les quatre opérateurs). Si nous comptons le fait que nous partirons de 0, il faut que nous soyons capables de conserver en binaire des valeurs allant de 0 à 13 (ce qui fait bien 14 symboles). Or, en binaire, 13 vaut **1101** ( $1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$ ). Il nous faut donc 4 bits pour représenter chacun de nos symboles :

```
0000 0
0001 1
0010 2
0011 3
0100 4
0101 5
0110 6
0111 7
1000 8
1001 9
1010 +
1011 -
1100 /
1101 %
1110 vide
1111 vide
```

Nous avons fixé le nombre de symboles d'une équation à 12. Ce choix n'était pas dû au hasard : chaque symbole est codé sur 4 bits, donc 12 symboles tiennent sur  $12 \times 4 = 48$  bits, soit 6 octets. Les chromosomes (qui représentent une solution au problème) seront donc codés sur 6 octets (chacun d'eux contenant deux gènes). Pour la lisibilité du code et par souci de simplification, nous utiliserons ici des chaînes de caractères pour coder les gènes. J'attire votre attention sur la consommation mémoire que cela va engendrer : un caractère est codé sur un ou plusieurs octet(s) suivant les langages et donc, pour chaque information nous consommons au minimum 50% de mémoire en plus. De même, ce type d'algorithme se prête particulièrement bien à la programmation orientée objet (Gene, Chromosome, Population), mais la résolution sera présentée sous forme de programmation impérative, de manière à ne pas polluer la compréhension de l'algorithme par des considérations techniques.

Pour créer notre population initiale **P<sub>0</sub>**, nous allons devoir générer de manière aléatoire des chromosomes, puis les enregistrer dans une liste (tableau).

DÉVELOPPEZ  
VOS COMPÉTENCES  
EN OPEN SOURCE !

**Linagora Formation, c'est :**  
100 cycles de formations  
5 sites dédiés en Europe  
leader de formation LPI en France

## NOS SESSIONS EN MAI

<b>PARIS</b>	LPI 303	12 au 16
	LPI 102	19 au 22
<b>TOULOUSE</b>	LPI 101	12 au 15
<b>BORDEAUX</b>	LPI 101	19 au 22

## NOS SESSIONS EN JUIN

<b>PARIS</b>	LPI 201	10 au 13
	LPI 202	16 au 19
	Nagios	2 au 6
	Apache	16 au 18
<b>TOULOUSE</b>	LPI 102	10 au 13
<b>BORDEAUX</b>	LPI 201	23 au 26
<b>METZ</b>	LPI 101	16 au 19
	LPI 102	23 au 26

[www.formation.linagora.com](http://www.formation.linagora.com)

L'algorithme est très simple ; sachant que `nombre_genes` contient le nombre de gènes, `taille_gene` représente la taille d'un gène en bits, et la fonction `random()` renvoie `0` ou `1` de manière aléatoire :

```
chromosome <- ""
Pour i variant de 1 à nombre_genes Faire
    Pour j variant de 1 à taille_gene Faire
        chromosome <- chromosome + random()
```

Le code suivant réalise cette opération et affiche les chromosomes de la population générée.

```
01: # -*- coding: utf-8 -*-
02:
03: import random
04:
05: def chromosome_initial(nb_genes=12, taille_gene=4):
06:     chromosome = ""
07:     for i in range(nb_genes):
08:         for j in range(taille_gene):
09:             chromosome += str(random.randint(0, 1))
10:     return chromosome
11:
12:
13: def population_initiale(individus=100):
14:     population = []
15:     for i in range(individus):
16:         population.append(chromosome_initial())
17:     return population
18:
19:
20: if __name__ == "__main__":
21:     P0 = population_initiale()
22:     for chromosome in P0:
23:         print(chromosome)
```

Le paramétrage par défaut des fonctions (lignes 5 et 13) permet de traiter notre problème, mais permet également d'apporter des modifications par la suite (affinage des paramètres ou récupération du code). Nous obtenons des lignes semblables à la suivante :

```
0111110100110010101101010010011100000000
```

Il y a bien 48 caractères, donc visuellement, notre programme fonctionne correctement... Mais ce n'est pas très lisible ! Je vous rappelle que ce chromosome est censé représenter une solution potentielle à notre problème. Difficile de dire ici si c'est le cas ou non. Il nous faut donc une fonction de conversion capable de lire gène par gène un chromosome et de nous renvoyer sa représentation symbolique. Considérons que nous stockons les symboles dans un tableau dont les indices correspondent au codage binaire :

```
codage = ("0", "1", ..., "9", "+", "-", "/", "%", "_", "_")
```

Si nous voulons savoir à quoi correspond le gène **1101**, il suffit de convertir la valeur en décimal (13) et de regarder la valeur stockée dans le tableau à cet indice (`codage[13]` contient `%`). Pour que ce système fonctionne, il ne faut surtout pas oublier que les valeurs **1110** et **1111** ne correspondent à rien, mais que des valeurs doivent leur être associées de manière à ne pas obtenir d'erreur en recherchant un élément inexistant. Ici, j'ai choisi d'indiquer par un `_` qu'il n'y avait pas de symbole associé (on aurait aussi pu choisir une chaîne vide pour ne pas générer d'erreur et autoriser des solutions plus petites en termes de nombre de symboles).

Voyons maintenant comment coder cela :

```
...
20: def gene_to_symbole(gene, codage):
21:     return codage[int(gene, 2)]
22:
23:
24: def chromosome_to_expression(chromosome, codage, taille_gene=4):
25:     expression = ""
26:     for num_gene in range(int(len(chromosome) / taille_gene - 1)):
27:         expression += \
28:             gene_to_symbole(chromosome[num_gene * 4:num_gene * 4 + 4], codage)
29:     return expression
30:
31:
32: if __name__ == "__main__":
33:     codage = tuple(map(str, range(10))) + ("+", "-", "/", "%", "_", "_")
34:     P0 = population_initiale()
35:     for chromosome in P0:
36:         print(chromosome)
37:         print(chromosome_to_expression(chromosome, codage))
```

La fonction `gene_to_symbole()` des lignes 20 et 21 effectue la conversion de binaire en décimal avec `int(gene, 2)`, puis nous retourne le symbole associé au gène dans le tableau de codage. La fonction `chromosome_to_expression()` des lignes 24 à 29 parcourt un chromosome qui lui est passé en paramètre, lit des blocs de quatre caractères (qui représentent les bits) et les transforme en symbole. Au fur et à mesure, elle construit l'expression complète portée par le chromosome. Dans le programme principal, j'ai utilisé une astuce en ligne 33 pour ne pas avoir à écrire à la main tous les éléments du tuple contenant la traduction des gènes : `map(str, range(10))` applique la fonction `str()` à tous les éléments de la liste `range(10)`, soit

**[0, 1, ..., 9]**. Partant d'une liste d'entiers, on obtient une liste de chaînes de caractères que l'on convertit ensuite en tuple (en Python, les tuples sont des tableaux non modifiables).

On voit maintenant immédiatement la correspondance entre notre chromosome et l'expression qu'il représente :

```
0111110100110010101101010010101000100111000000000
7%32-52+270
```

Nous savons coder et décoder les données contenues dans les chromosomes. Attaquons-nous maintenant à la sélection des individus et à leur reproduction.

### 3.3 La résolution

#### 3.3.1 Évaluation

Ici, nous allons devoir concevoir la fonction de fitness. Qu'est-ce qui peut refléter le fait que nous soyons plus proches ou éloignés de la solution (obtenir 1234) ? Soit **Sol** la solution proposée par un chromosome. Son évaluation va nous donner un résultat : si c'est 1234, nous avons trouvé le résultat et plus la valeur est éloignée de 1234, plus la solution est mauvaise. En analysant le résultat de **|1234 - Sol|**, plus la valeur est proche de **0**, meilleure est la solution. Certaines solutions vont générer des erreurs (présence de `_` ou division par `0`). Nous fixerons alors arbitrairement une valeur suffisamment grande pour indiquer que la solution n'est pas bonne.

En Python, tout cela donne :

```
33: def fitness(chromosome, codage, taille_gene=4, resultat_final=1234):
34:     expression = chromosome_to_expression(chromosome, codage,
35:                                            taille_gene)
36:     try:
37:         resultat = abs(resultat_final - eval(expression))
38:     except:
39:         resultat = sys.maxsize
40:     return resultat
```

En ligne 36, nous utilisons la fonction `eval()` pour calculer le résultat de l'expression contenue dans le chromosome. En cas d'erreur, nous récupérons l'exception et affectons à `resultat` la plus grande valeur qu'un entier puisse contenir (ligne 38). Bien sûr, il ne faut pas oublier d'importer le module `sys` en début de programme.

Si nous parcourons la population, nous pouvons associer à chaque individu un score :

```
42: def evaluer_population(population, codage, taille_gene=4, resultat_final=1234):
43:     for indice, chromosome in enumerate(population):
44:         population[indice] = (fitness(chromosome, codage, taille_gene,
45:                                       resultat_final), chromosome)
46:     population.sort()
```

Comme nous travaillons sur une liste, il est inutile de renvoyer une valeur, car elle est directement modifiée en mémoire. Nous modifions chaque élément de la liste `population` en associant à chaque chromosome le résultat de la fonction `fitness()`. La liste de population est alors du type `[(fitness(c_1), c_1), ... (fitness(c_n), c_n)]` où `c_i` représente le *i*ème chromosome (`i` compris entre `1` et `n`). La dernière instruction de la ligne 46 permet de trier les individus de la population de manière à ce que ceux possédant le score de fitness le plus faible soient les premiers. Cette nouvelle représentation va nous amener à modifier la génération de la population initiale et du coup, la fonction d'évaluation :

```
...
14: def population_initiale(individus=100):
15:     population = []
16:     for i in range(individus):
17:         population.append((None, chromosome_initial()))
18:     return population
...
42: def evaluer_population(population, codage, taille_gene=4, resultat_final=1234):
43:     for indice, (value, chromosome) in enumerate(population):
44:         population[indice] = (fitness(chromosome, codage, taille_gene,
45:                                       resultat_final), chromosome)
46:     population.sort()
```

#### 3.3.2 Sélection

Pour la sélection, nous adopterons la technique la plus simple avec une sélection élitiste. Par défaut, nous conserverons la moitié de la population contentant les meilleurs candidats à une solution.

```
49: def selection(population, pourcentage=0.5):
50:     return population[0:int(len(population) * pourcentage)]
```

Pour voir les individus sélectionnés dans la population initiale :

```

52: if __name__ == "__main__":
...
54:     P0 = population_initiale()
...
58:     evaluer_population(P0, codage)
59:     P0 = selection(P0)
60:     print(P0)

```

### 3.3.3 Reproduction

Les individus étant sélectionnés, il faut qu'ils puissent se reproduire. Pour cela, nous devons créer des fonctions qui permettent de mettre en place les recombinations génétiques.

Commençons par le crossing-over : il nous faut deux individus, les parents, qui vont donner naissance à deux autres individus, les enfants.

```

54: def crossingover(parent_1, parent_2):
55:     chromosome_1 = parent_1[1]
56:     chromosome_2 = parent_2[1]
57:     point_croisement = random.randint(1, len(chromosome_1) - 2)
58:     enfant_1 = (None, chromosome_1[0:point_croisement] +
59:                 chromosome_2[point_croisement:])
60:     enfant_2 = (None, chromosome_2[0:point_croisement] +
61:                 chromosome_1[point_croisement:])
62:     return [enfant_1, enfant_2]

```

On détermine de manière aléatoire un point de croisement en ligne 54. Ensuite, il faut découper le chromosome de chaque parent suivant ce point et les coller en les mélangeant (lignes 55 à 58). Ici, le crossing-over est volontairement simple, tel qu'expliqué dans la partie traitant de la biologie... Mais dans la nature, il existe des crossing-over avec points de croisement multiples, que nous aurions pu également implémenter ici, de manière à brasser encore plus l'information génétique. Cela entraîne l'apparition d'un paramétrage supplémentaire avec le taux de crossing-over à un point de croisement, deux ou trois, sachant bien sûr que plus le nombre de points de croisement augmente, plus la probabilité d'apparition est faible. Nous restons donc ici sur le cas le plus simple.

Pour les mutations, nous appliquerons des substitutions : de manière aléatoire, nous sélectionnons une base (un bit du chromosome) et nous l'inversons (s'il vaut 1 il passe à 0 et inversement).

```

65: def mutation(individu):
66:     chromosome = individu[1]
67:     base = random.randint(0, len(chromosome) - 1)
68:     mute = "1" if chromosome[base] == "0" else "1"
69:     chromosome = chromosome[0:base] + mute + chromosome[base + 1:]
70:     return (None, chromosome)

```

Nous pouvons créer de la « nouveauté » à partir d'une population existante. Il faut maintenant produire les nouvelles générations.

### 3.3.4 Production des nouvelles générations

Pour produire une nouvelle génération, nous allons partir d'une population à laquelle nous allons appliquer notre fonction de sélection, puis reproduire les « meilleurs » individus avec crossing-over et mutations suivant certains taux. Il s'agit clairement d'eugénisme et un tel concept sorti de son contexte d'algorithme informatique est assez effrayant.

Voici la fonction de production d'une génération **Pi+1** à partir d'une génération **Pi**.

```

073: def nouvelle_generation(population, codage, taille_gene=4, resultat_final=1234,
074:                           taux_selection=0.5, taux_crossingover=0.9,
075:                           taux_mutation=0.001):
076:     evaluer_population(population, codage, taille_gene, resultat_final)
077:     population = selection(population, taux_selection)
078:     meilleur = population[0][0]
079:     meilleur_eval = chromosome_to_expression(population[0][1], codage)
080:     # Mélange de la population sélectionnée
081:     random.shuffle(population)
082:     nouvelle_population = []
083:     # Crossing-over
084:     indice = 0
085:     taille_population = len(population)
086:     while (indice + 1) < taille_population:
087:         parent_1 = population[indice]
088:         parent_2 = population[indice + 1]
089:         indice += 2
090:         if random.random() < taux_crossingover:
091:             for i in range(int(1 / taux_selection)):
092:                 nouvelle_population += crossingover(parent_1, parent_2)
093:         else:
094:             for i in range(int(1 / taux_selection)):
095:                 nouvelle_population.append(parent_1)
096:                 nouvelle_population.append(parent_2)
097:     # Mutation
098:     for individu in nouvelle_population:
099:         if random.random() < taux_mutation:
100:             nouvelle_population[indice] = mutation(individu)
101:     return (meilleur, meilleur_eval, nouvelle_population)

```

Nous utilisons ici toutes les fonctions définies précédemment. Ce qu'il faut surtout noter, ce sont les boucles des lignes 91 et 94 qui permettent de s'assurer que la population compte toujours le même nombre d'individus : une diminution de ce nombre entraînant une chute de la diversité génétique, notre algorithme s'arrêterait rapidement sans solution. Le crossing-over et la

mutation sont soumis à des taux d'apparition (lignes 90 et 99), ce qui permet d'effectuer des réglages en fonction des résultats observés.

### 3.3.5 Recherche d'une solution

Le programme principal va consister à créer de nouvelles générations jusqu'à obtenir un résultat à notre problème. Comme nous savons qu'il est possible que l'on ne trouve qu'une solution approchée, nous arrêterons notre programme pour le relancer avec une nouvelle population initiale à partir de la 3000ème génération.

```

104: if __name__ == "__main__":
105:     codage = tuple(map(str, range(10))) + ("+", "-", "/", "%", "_", "_")
106:     n_generation = 0
107:     P0 = population_initiale()
108:     while True:
109:         meilleur_score, meilleur_evaluation, P1 = nouvelle_generation(P0,
110:                                                                       codage)
111:         P0 = P1
112:         n_generation += 1
113:         print("Génération: {} Meilleur score: {}".format(
114:             n_generation, meilleur_score))
115:         print(" {}".format(meilleur_evaluation))
116:         if meilleur_score == 0:
117:             break
118:         if n_generation == 3000:
119:             print("Il vaut mieux repartir sur une nouvelle population...")
120:             evaluer_population(P1, codage)
121:             print(P1)
122:             P0 = population_initiale()
123:             n_generation = 0
124:             print("Appuyez sur <Return> pour continuer")
125:             input()

```

Lors du lancement du programme, si vous êtes chanceux, vous devriez obtenir rapidement une solution :

```

Génération: 1 Meilleur score: 1164
73+4-6-1%80
...
Génération: 26 Meilleur score: 0
374-37++897

```

Il arrive que le programme n'arrive plus à générer de nouvelles solutions. C'est là que nous arrêtons les calculs pour recommencer avec une nouvelle population initiale. Si vous observez la population à cet instant, vous vous apercevez qu'il n'y a plus de diversité génétique. Une solution pourrait consister à en réintroduire en augmentant le taux de mutations. Vous pouvez

également essayer d'ajuster les paramètres de l'algorithme, mais attention : un taux de mutations trop élevé ferait perdre le bénéfice de la sélection !

## Conclusion

Nous avons pu voir une catégorie d'algorithmes bien particuliers, puisque recherchant une solution à tâtons et s'appuyant sur un modèle biologique. Lors de leur utilisation, les problèmes principaux viendront du choix de la fonction de fitness, de la méthode de sélection et du paramétrage employé : chacun de ces éléments peut avoir un impact très important sur la rapidité de l'algorithme à converger vers une solution. De plus, ces réglages ne sont pas généraux et devront donc être effectués pour chaque problème. Dans la dernière partie, j'ai tenu à vous présenter un code mettant en œuvre les algorithmes génétiques sur un problème simple. Ce code n'est pas optimal, mais il a, je l'espère, le mérite d'être clair. Si vous souhaitez vous entraîner sur un autre problème relativement simple, vous pouvez tester la découverte d'un chemin dans un labyrinthe. Sinon, il ne vous reste plus qu'à trouver un problème sur lequel appliquer les algorithmes génétiques... ■

## Références

- [1] Eiben A. et J. Smith, « Introduction to evolutionary computing », éd. Springer, 2003.
- [2] Holland J., « Adaptation in Natural and Artificial Systems », University of Michigan Press, 1975.
- [3] Darwin C., « Origin of species », 1859.
- [4] Mendel G., « Experiments in plant hybridization », 1865.
- [5] Haldane J., « The causes of evolution », 1932.
- [6] Mayr E., « Systematics and the origin of species, from the viewpoint of a zoologist », Harvard University Press, 1942.
- [7] Watson J. et Crick F., « Molecular structure of nucleic acids. A structure for deoxyribose nucleic acid », Nature, vol. 171, 1953, p. 737 et 738.
- [8] Backer J., « Reducing bias and inefficiency in the selection algorithm », Actes de Second International Conference on Genetic Algorithms and their applications, 1987, p. 14 à 21.
- [9] Miller B. et Goldberg D., « Genetic Algorithms, Tournament Selection, and the effects of noise », Evolutionary Computation, vol. 4, p. 113 à 131.