Les algorithmes génétiques et leur formalisme mathématique.

Quels sont les principes mathématiques gouvernant les algorithmes génétiques et leurs applications ?

Mathématiques & Informatique
Yvann Le Fay, Killian Henger

Septembre 2016

Figure 1: Portrait de Darwin recomposé par un AG [1]

1 Preface

Les algorithmes génétiques ont été développés dès les années 1960 par John H. Holland, professeur de psychologie et de génie informatique à l'université du Michigan, ils interviennent dans des problèmes d'optimisation et ont pour fondement le croisement, la mutation et la sélection élitiste, des phénomènes dérivés des mécanismes d'évolution naturels.

Tout au long de ce document, nous essaierons de définir ces algorithmes mathématiquement puis nous proposerons plusieurs de leurs implémentations possibles.

Sommaire

2 Idées basiques et concept

2.1 Introduction

La première étape pour résoudre un problème mathématiquement est de modéliser le système avec des restrictions, idéalisations et simplifications, puis appliquer les outils mathématiques pour résoudre celui-ci et enfin en tirer des conclusions sur la problématique.

Depuis plus de 60 ans, des paradigmes sont rentrés en jeu, en un certain sens, de nouvelles méthodes permettant de répondre à nos problèmes d'une nouvelle façon, ceci étant possible par l'amélioration de la puissance de calcul de nos machines. En parallèle, on peut très bien observer que la nature s'en est toujours très bien sorti sans même que le modélisme mathématique soit développé par une quelconque conscience supérieure. Plus précisément, il y a un très grand nombre de processus et de mécanismes sophistiqués dans notre monde qui ont toujours intéressé les chercheurs par leur perfection admirable. Ils les ont alors imités en les modélisant mathématiquement pour certains, afin de mieux les comprendre et peut-être même les appliquer pour répondre à des besoins. Pour citer quelques-unes des technologies inspirées par la nature : éoliennes basées sur les nageoires de baleine, des plumes de paon pour des écrans, papillons de nuit aux yeux anti-reflets, le velcro, etc. Les méthodes informatiques quant à elles, utilisées pour modéliser ces systèmes en les analysant se sont montrées très efficaces dans énormément de disciplines, voici brièvement 3 de ces méthodes :

Réseaux neuronaux artificiels (ANNs) et convolutifs (ConvNet) : Des modèles simples de neurones et la manière dont ils intéragissent ensemble; ils sont surtout utilisés pour approcher des fonctions inconnues, pour de l'apprentissage-machine (langages, mathématiques, jeux (échecs, Go, etc)), traitement de signal, simulation (prévisions météorologiques, etc), optimisation (finances, gestion, régulation de trafic, etc), classification, etc.

Recuit simulé (Simulated Annealing) : Méthode d'optimisation probabiliste inspirée d'un processus en métallurgie.

Logique floue (Fuzzy Control) : Processus pour lequel on a aucun modèle analytique disponible et pourtant, on peut modéliser celui-ci par des modèles mathématiques incertains en utilisant des notations linguistiques, des règles et ensembles flous. Cette méthode est incertaine et est une forme de raisonnement par approchement.

Une quatrième méthode est notre sujet : les algorithmes génétiques (AGs, GAs).

Le monde comme on le vit aujourd'hui, avec ses multitudes de créatures toutes différentes des unes des autres, des individus extrêmement bien adaptés à leur environnement, en harmonie avec une balance écologique, est le produit de trois milliards d'années d'évolution, un processus basé sur la reproduction, la sélection naturelle et la mutation. La complexité et la capacité d'adaptation de ces créatures ont été acquises à travers la redéfinition et la combinaison constante d'informations génétiques sur une période très longue.

En bref, les algorithmes génétiques sont des simulations de notre évolution. Bien sûr dans la plupart des cas ce ne sont que des optimisations probabilistes basées sur les principes simples de l'évolution.

L'idée apparaît pour la première fois dans la thèse de doctorat de J.D. Bagley en 1967, "The Behavior of Adaptive Systems Which Employ Genetic and Correlative Algorithms"[2]. La théorie et les applications ont été décrites par John H. Holland, considéré comme le pionnier de ce domaine[3][4].

2.2 Définition et structure

Comme première approche, on peut considérer les AGs comme de simples algorithmes d'optimisation qui ont pour but de trouver les solutions dont l'indice de fitness est le plus élevé (ou le plus bas selon l'expression de celui-ci, maximum pour la suite du document).
En général, les problèmes d'optimisation sont formalisés sous cette forme [5] :

Trouver un x_0 tel que :

$$x_0 \in X | f(x_0) = \max_{x \in X} f(x). \tag{2.1}$$

Dans certains cas, il est quasi-impossible d'obtenir des solutions qui remplissent la condition (2.1) au sens strict. Cela dépend forcément du problème, un maximum local ou une valeur proche du maximum local ou global peut convenir.

Assumons qu'on recherche au moins une valeur x où la fonction f est "aussi haute que possible".

L'espace X est une analogie de l'ensemble de compétition des individus (monde réel ou scolaire par exemple), alors f assigne une valeur objective au "fitness" de chaque individu pour un problème précis (valeur objective bien que le choix de f soit laissé à l'utilisateur).

En parallèle avec le monde réel, la reproduction et l'adaptation sont portées sur le code génétique. Par conséquent, les algorithmes génétiques n'effectueront pas d'opération sur les individus directs de X mais sur une version encodée d'eux (qu'on appelle string pour chaîne de caractères).

2.2.1 Définition.

Soit S un ensemble de strings, X l'espace de recherche pour l'optimisation du problème comme formulé précédemment, alors une fonction :

$$\begin{aligned} c : X &\longrightarrow S \\ x &\longmapsto c(x) \end{aligned}$$

est une fonction dîte *fonction d'encodage*. Inversement, une fonction :

$$\begin{aligned} \tilde{c} : S &\longrightarrow X \\ s &\longmapsto \tilde{c}(s) \end{aligned}$$

est appelée *fonction de décodage*. Les fonctions d'encodage et de décodage, qui doivent être spécifiées selon le problème, ne sont pas forcément bijectives. Toutefois, comme en cryptographie, il est utile de travailler avec des fonctions injectives, l'égalité suivante est supposée être vérifiée (bijection réciproque) :

$$(c \circ \tilde{c}) \equiv \mathrm{Id}_S, \tag{2.2}$$

ce qui peut aussi s'écrire :

$$c(\tilde{c}(S)) = \tilde{c}(c(S)).$$

Finalement, la formulation générale du problème encodé de maximisation s'écrit :

Trouver un $s_0 \in S$ tel que $\tilde{f} = f \circ \tilde{c}$ est aussi haut que possible.

Autrement dit, Trouver un s_0 tel que :

$$s_0 \in S | f(\tilde{c}(s_0)) = \max_{\tilde{c}(s)} f(\tilde{c}(s)). \tag{2.3}$$

La table suivante liste différentes expressions utilisées en génétique ainsi que leur équivalent pour les algorithmes génétiques :

Génétique	Algorithme génétique
Genotype	String encodée
Phenotype	Point non codé
Chromosome	String
Gène	Position de la string
Allèle	Valeur à une certaine positon
Fitness	Indice d'approche d'une solution à un problème, de scorage

Maintenant, on peut écrire la structure d'un algorithme génétique en pseudo code.

2.2.2 Structure d'un algorithme génétique

$t := 0$;
Calcul ou génération de la population initiale $\mathcal{P}_0$;
while *conditions d'arrêt non remplies* **do**
    Sélections des individus pour la reproduction dans $\mathcal{P}_t$;
    Créations des descendants en croisant les individus sélectionnés;
    Mutations éventuelles de certains individus;
    Calcul de la nouvelle population $\mathcal{P}_{t+1}$;
end

Définissons les quatres processus de transition d'une génération à la prochaine génération.

Sélection : Mécanisme de sélection d'individus (strings) pour la reproduction selon leur indice de fitness (fonction de scorage).

Croisement : Méthode de fusionnement des informations génétiques de deux individus; l'encodage joue un rôle clé, en effet si celui-ci a correctement été choisi, deux bons parents produisent dans la plupart des cas un bon enfant.

Mutation : Dans la nature, le matériel génétique peut être modifié par des reproductions erronées aléatoires ou par la déformation des gènes, e.g. rayons gammas. Concernant les AGs, une mutation est réalisée comme une déformation "aléatoire" des strings, une certaine probabilité est associée à cette déformation. L'effet positif est la préservation de la diversité génétique afin d'éviter les maximums locaux ou encore les boucles.

Échantillonnage : Procédure qui calcule la nouvelle génération à partir de la précédente et de ses descendants.

Il est important de noter les principales différences entre les algorithmes génétiques et les méthodes d'optimisation traditionnelles (comme celle de Newton) :

1. Les AGs manipulent des paramètres encodés plutôt que les paramètres en eux-même, i.e. l'espace de recherche est S plutôt que X lui-même.

2. Les AGs appliquent des opérateurs de transition pour lesquels des probabilités sont associées, autrement dit une nouvelle génération est calculée à partir de l'ancienne mais aussi à partir de paramètres "aléatoires" (mutation). Ces opérateurs aléatoires offrent donc la possibilité de fournir des solutions nouvelles (au sens où elles n'étaient pas facilement prévisibles à l'aide de raisonnements mathématiques) à un problème. Les méthodes conventionnelles, elles, appliquent des opérateurs de transition purement déterministes.

3. Les AGs n'utilisent normalement aucune autre information concernant la fonction objective du problème (par exemple celle qu'un AG doit approcher) telle que les dérivées à l'ordre n de celle-ci. Ils peuvent donc être appliqués à un grand nombre de types de problèmes d'optimisations : continus ou discrets (optimisation combinatoire e.g. : $\sum_{s \in S} f(s) = f(S)$ [6]).

4. L'une des caractéristiques qui fait la robustesse des AGs est le fait qu'ils appliquent des opérations sur une population (ensemble de points (strings)) alors que la plupart des méthodes conventionnelles traitent point par point à chaque itération, i.e, la taille de la population est en quelque sorte 1. Cela augmente largement les chances d'obtenir un optimum global et donc vice-versa réduit les probabilités d'être bloqué dans un optimum global.

3 Un type d'AG simple

L'une des sous-classes des AGs les plus importantes est celle qui a comme entrée un nombre fixé de génotypes (ou strings encodées) en base 2. On présentera la théorie de ce type d'algorithmes, puis plus tard dans le document, des applications beaucoup plus concrètes. Pour cela, on affirmera que les strings encodées appartiennent toutes à l'ensemble :

$$S = \{0,1\}^n ,$$

où n est donc la longueur de ces strings. La taille de la population, autrement dit le nombre de strings de la population, sera noté $m = \text{Card}(\mathcal{P}_t)$. La génération au temps t est donc une liste de m strings, on la note :

$$\mathcal{P}_t = (b_{1,t}, ..., b_{m,t}).$$

3.1 Structure de ce type AG

Les AGs abordés dans cette partie peuvent être écrits sous cette forme :

$t := 0$;
Calculer ou générer la population initiale $\mathcal{P}_0$;
while *conditions d'arrêt non remplies* **do**
    for $i := 1$ **to** m **do**
        Sélection d'un individu $b_{i,t+1}$ de $\mathcal{P}_t$;
    end
    for $i := 1$ **to** $m - 1$ **step** 2 **do**
        if Rnd$[0,1] < p_k$ **then**
            Croisement $b_{i,t+1}$ avec $b_{i+1,t+1}$;
        end
    end
    for $i := 1$ **to** m **do**
        Mutation éventuelle de $b_{i,t+1}$;
    end
    $t := t + 1$;
end

Ici, la sélection, le croisement (avec une condition dépendant d'une variable p_k (rôle d'une probabilité)) et la mutation sont des degrés de liberté alors que l'échantillonnage est déjà spécifié. Chaque individu sélectionné est remplacé par un de ses enfants après croisement et mutation; ceux non sélectionnés "meurent". C'est le type d'échantillonnage le plus commun pour cette classe d'AG, d'autres variantes existent et proposent pour certaines, une meilleure efficacité.

3.2 Opérations génétiques sur les strings binaires

3.2.1 Sélection

La sélection est le procédé qui guide un algorithme génétique vers la solution et ce, en favorisant les individus à hauts fitness, défavorisant ceux qui ont des scores bas. Il peut être entièrement déterministe, mais dans la plupart des implémentations (notamment celles qu'on étudiera plus tard), un composé est "aléatoire". Le moyen le plus populaire d'effectuer cette sélection et dont les propriétés avantageuses ont été démontrées, est le suivant : la probabilité de choisir un individu précis est proportionnelle à l'indice de fitness associé à cet individu, cela peut être modélisé sous la forme d'une expérience aléatoire avec donc l'égalité suivante :

$$\Pr\left[\mathrm{Sel}(b_{j,t}) = 1\right] = \frac{f(b_{j,t})}{\sum_{k=1}^{m} f(b_{k,t})}, \tag{3.1}$$

avec, à un moment j de la sélection t, l'expression booléenne suivante :

$$\mathrm{Sel}(b_{j,t}) = \begin{cases} 0 & b_{j,t} \text{ n'a pas été sélectionné dans } \mathcal{P}_t \text{ pour } \mathcal{P}_{t+1}. \\ 1 & b_{j,t} \text{ a été sélectionné dans } \mathcal{P}_t \text{ pour } \mathcal{P}_{t+1}. \end{cases}$$

Pour modéliser l'opération de sélection sous forme d'expérience aléatoire, il faut que nos fitness soient toutes positives, si cela n'est pas le cas, on peut utiliser une transformation ϕ croissante et telle que :

$$\phi(f(b_{k,t})) : \mathbb{R} \mapsto \mathbb{R}^+.$$

Dans le cas le plus simple, on peut définir ϕ comme une fonction de décalage, soit F_t l'ensemble des fitness de nos m individus à la génération t :

$$F_t = \bigcup_{j=1}^{m} f(b_{j,t}).$$

Notre fonction de décalage est donc :

$$\phi(f(b_{k,t})) = f(b_{k,t}) + |\min_{F_t} f(b_{j,t})|.$$

On a alors la propriété suivante qui est respectée :

$$\min_{F'_t} \phi(f(b_{j,t})) \geq 0$$

Ainsi (3.1) devient :

$$\Pr\left[\mathrm{Sel}(b_{j,t}) = 1\right] = \frac{\phi(f(b_{j,t}))}{\sum_{k=1}^{m} \phi(f(b_{k,t})} . \tag{3.2}$$

On peut forcer la propriété (3.1) à être remplie en appliquant une expérience aléatoire (on la représente souvent comme un jeu de roulette dont les emplacements ne sont pas répartis de manière égale, i.e les différentes sorties peuvent se produire mais avec des probabilités différentes). La formulation algorithmique du système de sélection (3.1) peut être écrit sous la forme suivante :

Algorithme.

$i := 1$;
$x := \text{Rnd}[0, 1]$;
while $i < m \wedge x < \sum_{j=1}^{i} f(b_{j,t}) / \sum_{j=1}^{m} f(b_{j,t})$ **do**
  $i := i + 1$;
end
Sélectionner $b_{i,t}$;

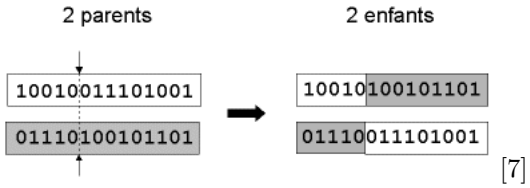
Par analogie pour (3.2), cette méthode de sélection est appelée sélection proportionnelle.

3.2.2 Croisement

Dans le cas de la reproduction comme elle apparaît dans la nature, le matériel génétique des deux parents est mélangé, cela a pour conséquence qu'une partie des gènes de l'enfant provient d'un parent et le reste de l'autre parent.

Ce mécanisme est appelé croisement (ou en anglais crossover), c'est un outil extrêmement puissant pour introduire de nouveaux matériaux génétiques et donc maintenir une certaine diversité. Cet outil doit avoir pour propriété que deux bons parents produisent dans la plupart des cas un bon enfant, voire meilleur que les parents.

Le croisement est l'échange de gènes entre deux chromosomes de deux parents. Dans le cas le plus simple, on peut réaliser ce processus en coupant deux strings à une position aléatoire et en les échangeant. Cette méthode est appelée croisement à un point (one-point crossover).



[7]

Figure 2: Représentation de la méthode 'croisement à un point'

Algorithme.

$pos := \text{Random}\{1, ..., n-1\}$;
for $i := 1$ **to** pos **do**
  $Enfant_1[i] := Parent_1[i]$;
  $Enfant_2[i] := Parent_2[i]$;
end
for $i := pos + 1$ **to** n **do**
  $Enfant_1[i] := Parent_2[i]$;
  $Enfant_2[i] := Parent_1[i]$;
end

Cette méthode simple est souvent utilisée par les AGs qui opèrent sur des strings binaires. Pour certains problèmes ou encodages différents, d'autres méthodes de croisement peuvent être utiles voir nécessaires. On ne mentionnera qu'une petite partie des techniques existantes ([8] et [9]).

Croisement à N points (N-point crossover) : Plutôt qu'un, on choisit N points de coupure, N est aléatoire dans la plupart des cas. Chaque section est alors interchangée de la même façon que pour la première méthode. La sélection la plus efficace pour les problèmes encodés sur un alphabet binaire est celle à deux points de coupures.

Croisement segmenté (segmented crossover) : Similaire presque en tout point à la méthode de croisement à N points, la seule différence est que le nombre de points de coupure peut varier lors des différentes générations des populations.

Croisement brouillé (shuffle crossover) : On applique une permutation aléatoire aux deux parents, puis un croisement à N points, enfin, on réapplique la permutation inverse à la sortie.

Croisement uniforme (uniform crossover) : Pour chaque position, on décide aléatoirement si à cette position de coupure, on échange la partie ou non.

3.2.3 Mutation

Le dernier ingrédient de nos algorithmes génétiques est la mutation - la déformation aléatoire des informations génétiques d'un individu par l'influence environnementale. Dans notre monde, la probabilité qu'un certain gène soit muté est quasi égale pour tous les gènes. On fera cette analogie pour une string donnée s, où p_M est la probabilité qu'un seul gène soit modifié :

Algorithme.

for $i := 1$ **to** n **do**
  if $\mathrm{Rnd}[0,1] < p_m$ **then**
    Inversion de $s[i]$;
  end
end

Il est clair que p_M doit être relativement petit pour éviter que l'algorithme ne fasse trop de mutations, sinon le comportement de celui-ci serait quasi chaotique et s'approcherait d'une recherche aléatoire de la solution.

Pareil, comme dans le cas du croisement, le choix des techniques de mutation dépend fondamentalement de l'encodage et du problème en lui-même, voir [8] et [9].

Inversion d'un byte : Avec la probabilité p_M, un byte choisi aléatoirement est inversé (xor).

Inversion totale des bytes : La string entière est inversée byte par byte avec la probabilité p_M.

Remplacement aléatoire : Avec la probabilité p_M, la string est remplacée (complètement ou partiellement) par une autre.

3.2.4 Récapitulation

Toutes les routines ayant été décrites, on peut écrire un algorithme génétique universel (pour les espaces de recherche binaires).

Algorithme.

$t := 0$;
Calculer ou générer la population initiale $\mathcal{P}_0$;
while *conditions d'arrêt non remplies* **do**
    (* sélection proportionnelle *)
        for $i := 1$ **to** m **do**
            $x := \text{Rnd}[0, 1]$;
            $k := 1$;
            while $k < m \wedge x < \sum_{j=1}^{k} f(b_{j,t}) / \sum_{j=1}^{m} f(b_{j,t})$ **do**
                $k := k + 1$;
            end
            $b_{i,t+1} := b_{k,t}$;
        end
    (* croisement à un point *)
        for $i := 1$ **to** $m - 1$ **step** 2 **do**
            if $\text{Rnd}[0, 1] \leq p_k$ **then**
                $pos := \text{Rnd}\{1, ..., n - 1\}$;
                for $k := pos + 1$ **to** n **do**
                    $\nu := b_{i,t+1}[k]$;
                    $b_{i,t+1}[k] := b_{i+1,t+1}[k]$;
                    $b_{i+1,t+1}[k] := \nu$;
                end
            end
        end
    (* mutation *)
        for $i := 1$ **to** m **do**
            for $k := 1$ **to** n **do**
                if $\text{Rnd}[0, 1] < p_m$ **then**
                    Inversion de $b_{i,t+1}[k]$;
                end
            end
        end
    $t := t + 1$;
end

4 Analyse mathématique d'un schéma

> "It is not the strongest of the species that survives, nor the most intelligent that survives. It is the one that is most adaptable to change"
>
> — Charles R. Darwin

Ce chapitre a pour but d'expliquer pourquoi les algorithmes génétiques sont de bons algorithmes d'optimisation. Le montrer d'une manière certaine à l'opposé des arguments de Darwin qui en un certain sens, disait que sa théorie était acceptable tant qu'on n'avait pas démontré le contraire.

Pour des méthodes d'optimisation déterministes telles que la méthode des gradients ou encore celle de Newton, il est quasi sûr d'obtenir des résultats convenables à notre problème dans un nombre connu d'itérations (de plus les conditions qui font qu'on n'aboutisse pas à une solution sont connues), déterminé par une certaine complexité algorithmique. Pour tout algorithme d'optimisation probabiliste, on ne peut pas déterminer une convergence algorithmique forte du fait du caractère imprévisible de l'algorithme. On comprend vite que les affirmations concernant une quelconque convergence de ces algorithmes ne peuvent que fournir des informations sur le comportement normal moyen de ceux-ci. Dans le cas précis des algorithmes génétiques, il y a plusieurs circonstances qui rendent la prévision du comportement de convergence encore plus difficile :

1. Une simple transition contient déjà trois routines faisant intervenir des processus probabilistes : sélection, croisement et mutation, cette structure est compliquée et un modèle mathématique qui déterminerait la convergence moyenne d'un AG l'est aussi.

2. Pour chacune de ses routines, il existe plusieurs variantes qui ont des impacts directs sur la vitesse de convergence.

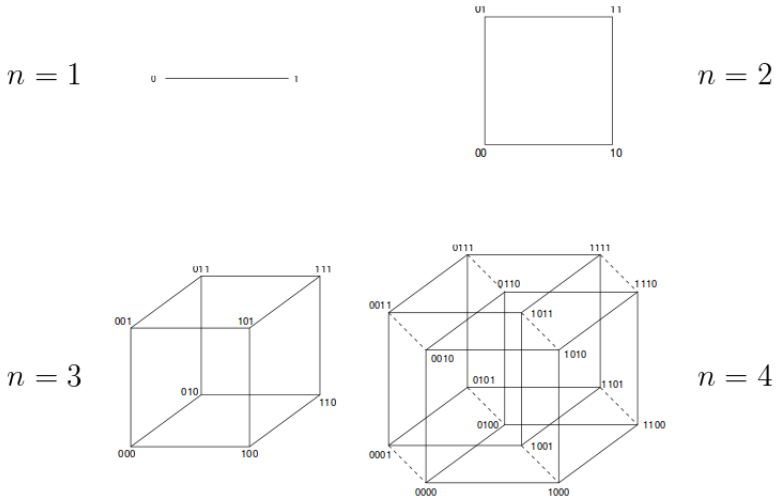
Par la suite, on va essayer de donner un théorème de convergence (bien que faible) pour les formes générales des AGs. Un théorème qui résumerait les résultats d'un AG et qui donnerait un élément de réponse à notre questionnement : pourquoi les algorithmes génétiques permettent de résoudre plusieurs classes de problèmes d'optimisation mais échouent dans certains cas (faisant même partie de ces classes) ? On se place dans le cas des AGs de type (3.1), i.e, des algorithmes génétiques de population de taille m, avec chaque individu de longueur n. Aucune autre supposition ne devrait être faite concernant les trois routines : sélection, croisement et mutation. Afin de tenter de trouver ce théorème, étudions la structure d'un AG de façon plus formelle.

Définition.

Une string $H = (h_1, ..., h_n)$ sur l'ensemble $\{0, 1, *\}$, alphabet, est appelé *schéma binaire* de longueur n. Les $h_i \neq *$ sont appelés *les spécifications* de H, un $h_i = *$ est appelé *joker*.

Un schéma binaire peut être considéré comme un sous-ensemble de $\{0, 1\}^n$ si l'on considère la fonction suivante qui est une bijection d'un schéma vers son sous-ensemble associé [10].

$$i : \{0, 1, *\}^n \to \mathcal{W}(\{0, 1\}^n)$$
$$H \mapsto \{S | \forall 1 \leq i \leq n : (h_i \neq *) \Longrightarrow (h_i = s_i)\}.$$



[1]

Figure 3: Hypercube de dimension 1-4, 2^n sommets (individus)

Si l'on interprète une string de longueur n par un hypercube de dimension n (Figure 3), alors un schéma peut être considéré comme une hypersurface appartenant à l'hypercube.

Avant d'introduire le théorème d'Holland, nous devons définir certaines notions concernant les schémas.

Définition.

1. Une string S sur l'alphabet $\{0, 1\}$ porte le schéma H si et seulement si H a les mêmes spécifications que S sur toutes ses positions qui ne sont pas des jokers, formellement :

   $$\forall i \in \{j | h_j \neq *\} : s_i = h_i,$$

   S est donc un sous-ensemble de H, $S \in H$.

2. Le *nombre de spécifications* d'un schéma H est appelé *ordre* et est noté :

   $$\mathcal{O}(H) = |\{i \in [\![1, n]\!] | h_i \neq *\}|.$$

3. La distance entre la première spécificaion et dernière spécification :

   $$\mathcal{L}(H) = \max\{i | h_i \neq *\} - \min\{i | h_i \neq *\},$$

   est appelée la *longueur de définition* d'un schéma H.

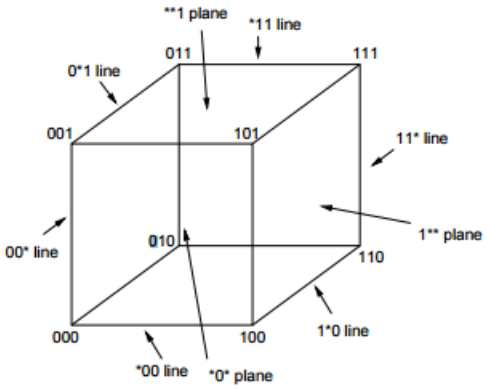


Figure 4: Une représentation "hyperplanère" d'un schéma pour $n = 3$ [11] [12]

4.1 Le théorème du Schéma d'Holland

On va aborder la formulation et la démonstration du théorème de convergence moyenne des algorithmes génétiques, le théorème du schéma d'Holland. Comme dit plutôt, celui-ci ne peut que donner une idée vague de la convergence de ces algorithmes, en comparaison avec ceux des méthodes d'optimisation conventionnelles.

On choisit une fonction de fitness arbitraire f, on note alors :

1. Le nombre d'individus qui remplissent le schéma H à un nombre d'étape t :

$$\psi_{H,t} = |\mathcal{P}_t \cap H| = \text{Card}(\mathcal{P}_t \cap H).$$

2. $\bar{f}(t)$, la simple moyenne arithmétique des fitness de nos individus à l'instant t :

$$\bar{f}(t) = \frac{1}{m}\sum_{i=1}^{m} f(b_{i,t}).$$

3. Le terme $\bar{f}(H,t)$ est la moyenne arithmétique des fitness de nos individus observées portant le schéma H au temps t :

$$\bar{f}(H,t) = \frac{1}{\psi_{H,t}} \sum_{i \in \{j | b_{j,t} \in H\}} f(b_{i,t}) = \frac{1}{\psi_{H,t}} \sum_{\substack{b_{j,t} \in H \\ j=1}}^{m} f(b_{i,t}).$$

4.1.1 Théorème (Théorème du Schéma - Holland 1975)

Supposons qu'on se place cette fois-ci dans le type plus précis (3.1;3.2.4)), alors l'inégalité suivante concernant l'espérance du nombre d'individus à la génération $t+1$ portant H est respectée pour tout schéma H :

$$\mathbb{E}[\psi_{H,t+1}] \geq \psi_{H,t} \frac{\bar{f}(H,t)}{\bar{f}(t)} \left(1 - p_k \frac{\mathcal{L}(H)}{n-1}\right)(1-p_m)^{\mathcal{O}(H)}. \tag{4.1}$$

Preuve. La probabilité qu'on sélectionne un individu qui est dans H est :

$$\Pr\left[\text{Sel}(b_{k,t} \in H) = 1\right] = \frac{\sum_{i \in \{j | b_{j,t} \in H\}} f(b_{i,t})}{\sum_{i=1}^{m} f(b_{i,t})}. \tag{4.2}$$

Il est important de noter que cette probabilité ne change pas lors de l'exécution de la routine de sélection. De plus, chaque individu des m individus est sélectionné de manière totalement indépendante des autres. De ce fait, le nombre attendu d'individus sélectionnés qui remplissent H est $m \Pr[\text{Sel}(b_{j,t} \in H) = 1]$ et on a :

$$m \Pr[\text{Sel}(b_{j,t} \in H) = 1] = \psi_{H,t} \frac{\bar{f}(H,t)}{\bar{f}(t)}.$$

Si deux individus remplissant H sont croisés, les deux sorties remplissent aussi H. Le nombre de strings qui remplissent H peut diminuer si et seulement si (dans ce type-ci) une string, qui remplit aussi H est croisée avec une string qui ne remplit pas H. Il est évident que ce n'est possible que dans le cas où l'on sélectionne des individus qu'on va croiser dans des ensembles qui ne remplissent pas des spécifications de H. La probabilité que l'ensemble de croisement est choisi dans la longueur de définition de H est

$$\frac{\mathcal{L}(H)}{n-1}.$$

La probabilité de survie p_S, i.e. la probabilité qu'une string remplissant H produit un enfant remplissant H, respecte forcément cette inégalité (on rappelle que le croisement est effectué qu'avec la probabilité p_k, on reconnait ici la forme de la probabilité de l'événement contraire exprimé au-dessus avec croisement effectué) :

$$p_S \geq 1 - p_k \frac{\mathcal{L}(H)}{n-1}.$$

Les routines de sélection et de croisement sont effectuées de manière aussi indépendante, donc on peut simplement faire le produit de nos deux relations précédentes pour donner une relation concernant le nombre prévu de strings remplissant H après croisement :

$$p_S \frac{\bar{f}(H,t)}{\bar{f}(t)} \psi_{H,t} \geq \frac{\bar{f}(H,t)}{\bar{f}(t)} \psi_{H,t} \left(1 - p_k \frac{\mathcal{L}(H)}{n-1}\right).$$

Après croisement, le nombre de string qui remplissent H ne peut que décroitre si une de ces strings est altérée par la mutation à une spécification précise de H. La probabilité que toutes les spécifications de H restent intactes par la mutation est donc l'événement contraire de : l'une des caractéristiques de H selon la probabilité p_M a été mutée, or il y a $\mathcal{O}(H)$ spécifications, la probabilité de cet événement est donc :

$$(1 - p_k)^{\mathcal{O}(H)}.$$

En appliquant le même argument, on peut encore faire le produit, on a donc la formulation sur l'espérance (4.1) qui suit.

L'argument qu'on a utilisé pour démontrer le théorème du schéma peut être appliqué de manière analogique à des algorithmes génétiques de différents types (différentes routines de croisement et de mutation).

Corolaire du théorème. *Pour les algorithmes génétiques de type 3.1 avec la sélection par roulette, l'inégalité suivante tient*

$$\mathbb{E}[\psi_{H,t+1}] \geq \frac{\bar{f}(H,t)}{\bar{f}(t)} \psi_{H,t} P_k(H) P_m(H), \tag{4.3}$$

toujours pour tout schéma H, où $P_k(H)$ est une constante dépendant seulement de H et de la méthode de croisement, $P_m(H)$ est une constante qui dépend de la même manière, seulement de H et de la routine de mutation de l'algorithme. Pour les variantes de (3.1.2) et (3.1.3), des estimations de ces constantes ont été fournies par [10] :

$$P_k(H) = 1 - p_k \frac{\mathcal{L}(H)}{n-1} \text{ croisement à un point}$$
$$P_k(H) = 1 - p_k \left(1 - \frac{1}{2^{\mathcal{O}(H)}}\right) \text{ croisement uniforme}$$
$$P_k(H) = 1 - p_k \text{ autre méthode de croisement}$$

$$P_m(H) = (1 - p_m)^{\mathcal{O}(H)} \text{ mutation totale des bytes}$$
$$P_m(H) = 1 - p_m \frac{\mathcal{O}(H)}{n} \text{ inversion d'un seul byte}$$
$$P_m(H) = 1 - p_m \text{ inversion totale des bytes}$$
$$P_m(H) = 1 - p_m \frac{|H|}{2^n} \text{ sélection aléatoire}$$

Le théorème du schéma d'Holland propose une réponse à l'explication des résultats de convergence des AGs d'un type différent à celui des explications de l'efficacité des algorithmes d'optimisation conventionnels. Une des implications du théorème du schéma est que les schémas avec des fitness au-dessus de la moyenne des fitness des individus et dont les définitions sont courtes, sans considérer les variances détaillées dans le corolaire, tendent à produire plus d'enfants "viables" que les autres. On peut donner un nom à ces schémas productifs, *blocs de construction.* Pourquoi les appeler blocs de construction ? Assumons que le quotient suivant :

$$\frac{\bar{f}(H,t)}{\bar{f}(t)},$$

est relativement constant au fil du temps et des générations, alors on a

$$\mathbb{E}[\psi_{H,t+n}] \geq \psi_{H,t} \left(\frac{\bar{f}(H,t)}{\bar{f}(t)} P_k(H) P_m(H)\right)^n.$$

ce qui revient à dire que le nombre de strings qui remplissent les schémas productifs, augmente de manière exponentielle (en fait comme une séquence géométrique), c'est la raison pour laquelle, ils sont appelés blocs de construction puisqu'ils tendent à construire de plus en plus de strings les remplissant.

On est en droit de se demander si c'est une bonne stratégie de laisser les schémas recevoir un nombre d'essais augmentant exponentiellement, si oui alors pourquoi (dans (4.1.2)) ? On peut se poser une autre question fondamentale en venant au fait que les AGs opèrent sur des strings binaires mais pas sur les schémas directement : le théorème du Schéma d'Holland ne fournit des observations que sur l'évolution des schémas selon leur fitness observée. Quelle est l'interprétation en pratique de ce comportement décrit par le théorème ?

Enfin, on peut se demander d'où vient le rôle crucial des blocs de constructions, et quelles influences ont la fonction de fitness et le schéma d'encodage sur le comportement global de l'algorithme.

4.1.2 L'allocation et la distribution optimale des essais

Le théorème pousse à penser que le nombre d'essais que les blocs de construction recoivent, ne fait que croître de manière exponentielle durant les générations. Est-ce une bonne stratégie que de laisser les blocs constructifs recevoir un nombre exponentiellement croissant d'essais ? D'après (11), ce problème est une analogie d'un problème bien étudié dans la théorie des décisions statistiques : *the two-armed bandit problem* et sa généralisation *the k-armed bandit problem.* Même si l'on peut penser que ce problème n'est pas tellement lié aux algorithmes génétiques, il est important, pour comprendre de manière profonde l'efficacité des algorithmes génétiques, de faire la connexion entre ce problème et le problème d'allocation et de la distribution optimale des essais.

La première analogie faite dans (11) est la suivante : supposons que l'on dispose d'une machine de jeu de hasard où l'on peut jouer à deux, c'est-à-dire qu'il y a deux commandes et deux emplacements pour insérer des jetons. On a toutefois, un seul joueur, il peut déposer un jeton soit dans l'emplacement de gauche soit dans l'emplacement de droit. Après avoir actionné la machine à partir de la commande du côté correspondant à l'emplacement où le joueur a glissé le jeton, le joueur perd soit sa mise soit il gagne et reçoit un certain prix. Pour des raisons de simplification, on ne travaille qu'avec les gains algébriques (i.e, ils peuvent être négatifs), c'est-à-dire la différence entre le gain brute (0 dans le cas où il a perdu) et la valeur de mise de départ. Disons que le bras gauche produit un gain avec une valeur moyenne μ_1 et une variance σ_1^2 alors que le bras droit produit un gain avec une valeur moyenne μ_2 et une variance σ_2^2. Sans perte de généralité, prenons $\mu_1 \leq \mu_2$. La question est donc, quel levier devrions-nous actionner pour optimiser nos gains ? On ne sait pas quel côté est associé aux plus grands gains moyens et on est donc face à un problème. Le joueur doit écrire les décisions qu'il prend tout en collectant des informations sur les gains, des informations qui lui permettraient de déterminer quel est le meilleur bras à jouer. Il y a donc deux parties, l'exploration (en jouant) et l'exploitation (en notant les résultats), on fait vite l'analogie avec les algorithmes génétiques.

Une approche simple de ce problème est de séparer l'exploration de l'exploi-tation. Plus spécifiquement, on peut effectuer une seule expérimentation (exploration) au départ, puis faire une décision irréversible basée sur le résultat de l'expérience. Supposons qu'on a N jetons. Si on alloue un nombre égal n (où $2n \leq N$) de jetons pour chaque bras afin de faire les expériences, on peut alors allouer les $N - 2n$ jetons restants aux bras présentants les meilleurs résultats. D'après [9], les pertes prévues sont données par :

$$L(N, n) = (\mu_1 - \mu_2)((N - n)q(n) + n(1 - q(n)))$$

avec $q(n)$, la probabilité que le pire bras observé est devenu le meilleur bras observé après $2n$ tours d'essai. L'idée pour comprendre cette expression est que si l'on observe que le pire bras est devenu le meilleur (probabilité $q(n)$), le nombre total d'essais alloués au bras droit est $N-n$. La perte est, donc, $(\mu_1 - \mu_2)(N - n)$. Dans le cas inverse, où l'on observe que le meilleur bras reste le meilleur, ce qui est donc associé à la probabilité $1 - q(n)$, la perte est maintenant $(\mu_1 - \mu_2)n$. On obtient alors l'expression de $L(N, n)$. Toujours d'après [9], le théorème central limite permet d'approcher $q(n)$ comme ci-suit :

$$q(n) \approx \frac{1}{\sqrt{2\pi}} \frac{e^{-c^2/2}}{c}, \qquad \text{avec } c = \frac{\mu_1 - \mu_2}{\sqrt{\sigma_1^2 + \sigma_2^2}} \sqrt{n}.$$

Pour répondre à notre question, on doit spécifier la taille raisonnable n de l'expérience. Si l'on choisit $n = 1$, l'information obtenue n'est que peu intéressante et l'échantillon n'est largement pas optimal. Si l'on choisit cette fois-ci, $n = \frac{N}{2}$, il ne nous reste plus d'essai possible pour utiliser les informations utilisées durant la phase de l'expérimentation. On doit donc trouver le n optimal pour "doser" la partie exploration et exploitation, presque aucune exploration ($n = 1$) et de l'exploration sans exploitation ($n = \frac{N}{2}$). On se doute que la manière optimale de séparer l'exploration de l'exploitation est quelque part vers le milieu de ces deux valeurs. Holland [4] a étudié ce problème et est arrivé à la conclusion que la stratégie optimale est donnée par l'équation

suivante :

$$n^* \approx b^2 \ln\left(\frac{N^2}{8\pi b^4 \ln(N^2)}\right), \qquad \text{avec } b = \frac{\sigma_1}{\mu_1 - \mu_2}.$$

Après quelques transformations, on obtient :

$$N - n^* \approx \sqrt{8\pi b^4 \ln(N^2)} e^{n^*/(2b^2)},$$

i.e que la stratégie optimale est d'allouer un peu plus qu'un nombre exponentiellement croissant d'essais au bras observé comme étant le meilleur. Même si dans la pratique du jeu, il n'est pas possible d'appliquer cette stratégie puisqu'il est nécessaire de connaître les valeurs moyennes μ_1 et μ_2. Holland a trouvé une limite importante dans la stratégie que l'on doit approcher pour obtenir un gain optimal. Dans le cas des algorithmes génétiques, cela implique que l'on doit fournir au moins un nombre exponentiellement croissant d'essais aux meilleurs blocs de construction observés. Pour autant, la connexion entre ce problème et les algorithmes génétiques peut ne pas être encore totalement claire. Considérons une position arbitraire dans une string. Il y a alors deux schémas d'ordre 1 qui ont leur seule spécification à cette position. Selon le théorème du Schéma d'Holland, l'algorithme génétique choisi implicitement un de ces deux schémas, pour se décider, seules des données incomplètes sont fournies sur les schémas (les fitness moyennes observées). En ce sens, l'algorithme génétique résout énormément de problemes "two-armed bandit" en parallèle.

Le théorème du schéma n'est pas restreint à des schémas d'ordre 1. En pratique, on observe qu'un algorithme génétique résout énormément de "k-armed bandits" problèmes en parallèle. Le problème des "k-armed bandit" est bien plus compliqué que celui du cas particulier "two-armed bandit", mais il se résout de manière analogique, toujours d'après [4] et [9], le nombre optimal d'essais à effectuer reste exponentiellement croissant.

4.1.3 Parallélisme implicite

Jusque-là, on a observé de deux manières différentes les algorithmes génétiques, deux manières qui sont à première vue en conflit :

1. Les AGs opèrent sur des strings.
2. Les AGs opèrent sur des schémas.

On est dans le droit de se demander sur quels objets les AGs opèrent-ils réellement, des strings ou des schémas ? La réponse est en fait qu'ils opèrent sur les deux en même temps. L'interprétation commune est que les AG opèrent implicitement sur un nombre important de schémas et cela est accompli en exploitant les données disponibles, des données incomplètes à propos de ces schémas ou de meilleurs schémas.

Cette propriété remarquable d'opérer sur les schémas en même temps que les exploiter, est appelée *parallélisme implicite* des AGs.

Un algorithme simple présenté dans le chapitre 3 ne traite que m structures en une étape, sans aucune mémorisation des autres générations. Combien de schémas un algorithme génétique traite-t-il réellement ? Une seule string binaire remplit n schéma d'ordre 1, $\binom{n}{2}$ schémas d'ordre 2, en général, $\binom{n}{k}$ schémas d'ordre k. Donc, une string remplit :

$$\sum_{k=1}^{n} \binom{n}{k} = 2^n$$

schémas. On obtient donc que pour n'importe quelle génération, il y a entre 2^n et $m2^n$ schémas, qui ont au moins une string les représentant. Combien parmi ces schémas sont réellement traités ? Holland [4] a donné une estimation de la quantité de schémas qui sont gardés pour la génération suivante. Ce résultat nous donne une information importante concernant la quantité large de schémas qui est traitée en parallèle, large même pour une quantité relativement petite de strings.

Théorème. *Considérons une génération de départ aléatoire d'un algorithme génétique simple (3.5) et laissons $\epsilon \in [0,1]$ être une marge d'erreur fixée. Alors un schéma de longueur*

$$l_s \leq \epsilon(n-1)+1$$

a une probabilité d'au moins $1-\epsilon$ de survivre au croisement à un point (à comparer à la preuve du théorème d'Holland). Si la taille de la population est choisie telle que $m = 2^{l_s/2}$, le nombre de schémas qui survivront pour la nouvelle génération est de l'ordre de $\mathcal{O}(m^3)$.

4.1.4 Théorème exact du schéma d'Holland

Dans la première partie abordant le théorème du schéma d'Holland, on a vu que celui-ci est le pilier de l'analyse théorique des algorithmes génétiques et pourtant, il présente deux grands défauts. En effet, il ne nous renseigne que sur une génération et ça en ne donnant qu'une approximation par minorant des fréquences d'apparition des schémas. Alden H. Wright dans [13] fournit la preuve d'un théorème plus exact en donnant une égalité concernant ces fréquences. Le théorème d'Holland n'est qu'une implication de celui-ci.

4.2 Blocs de construction et problème d'encodage

On a déjà introduit le terme de blocs constructifs pour un schéma avec une fitness moyenne importante et une longueur de définition relativement courte (en impliquant un ordre petit). Il serait intéressant d'expliquer pourquoi est-ce que cette notation est légitime. On a vu dans le théorème du schéma et dans (4.1.2) que les blocs constructifs reçoivent un nombre d'essais croissant de manière exponentielle. Les implications de (4.1.3) montrent qu'un certain nombre de schémas (inclus les blocs constructifs) sont évalués en parallèle et de manière implicite. Pour autant, cela ne fait pas le lien avec la performance, i.e la convergence des AGs. Malheureusement, il n'existe pas de théorie complète qui fournit une explication claire au lien, une hypothèse a tout de même été formulée.

4.2.1 Hypothèse des blocs constructifs

Un algorithme génétique créé pas à pas, de meilleures solutions à un problème donné en recombinant, en croisant et en mutant que peu, des schémas courts à haute fitness.

Dans [9], Golderberg donne une comparaison pour cette hypothèse :
Tout comme un enfant construit de magnifiques et puissantes forteresses à partir de petits bouts de bois, les algorithmes génétiques cherchent des performances optimales à travers la juxtaposition de courts et de performants schémas.

Cette affirmation semble raisonnable et correspond au théorème d'Holland (à rappeler l'exposant $\mathcal{O}(H)$ dans l'expression de l'inégalité). Est-ce que cette hypothèse est valide et si oui dans quelles conditions (on essaye ici de répondre à la question : "dans quel cas les algorithmes génétiques ne répondent pas au problème d'optimisation posé ?) ? Considérons une fonction fitness affine telle que :

$$f(s) = a + \sum_{i=1}^{n} c_i s[i],$$

la fonction de fitness ci-dessus est une combinaison linéaire de tous les gènes. Ici la valeur optimale peut être déterminée pour chaque gène indépendemment.

Cette fois-ci, considérons $f(x)$, une fonction telle que :

$$f(x) = \begin{cases} 1 & \text{si } x = x_0 \\ 0 & \text{sinon} \end{cases}$$

En conséquence de cette définition de notre fitness, il n'y a qu'une valeur optimale (x_0), pour tous les autres x, les fitness sont égales entre elles. De par ce fait, on ne considère pas le contenu de chaque x, la fonction de fitness ne guide plus l'AG et nous sommes en présence d'un simple algorithme de recherche (par ailleurs, on ne sait pas si le x calculé est x_0), la fitness n'a plus aucun intérêt si ce n'est que nous renseigner sur la vérité logique booléenne : x est-il optimal ? Dans ce cas, la probabilité d'obtenir $x = x_0$ est $\frac{1}{\text{Card}(X)}$. Il est clair qu'il est nécessaire de balayer tout X pour pouvoir calculer $f(x = x_0)$ et de par ce fait l'expression même de la probabilité d'obtenir x_0 du premier coup ne présente pas d'intérêt, si ce n'est que pour nous donner une idée de la difficulté à trouver une solution de cette manière.

Notre hypothèse de départ selon laquelle les blocs de construction mènent vers des solutions à nos problèmes plus rapidement est vrai lorsque la fonction de fitness dépend des gènes de nos strings, dans le second cas, il est très clair que puisque notre fonction de fitness ne nous renseigne pas sur la qualité de nos strings alors notre hypothèse ne peut s'appliquer. Il devient clair que plus l'on décompose nos strings pour composer notre fonction de fitness plus l'on guide rapidement l'AG vers la bonne solution, $f(x)$ dans le second cas, englobe l'ensemble des $s[i]$ de nos x, en considérant le tout. En soi plus l'on tient compte de la position relative de nos $s[i]$ dans nos strings, plus il est complexe pour l'AG d'utiliser cette information, à l'inverse, plus l'on considère de manière indépendante les gènes de nos strings plus la fitness guidera l'AG vers la bonne solution et cela vaut dans certaines mesures, pour tout algorithme d'optimisation.

Les biologistes ont nommé ce phénomène l'épistasie, on dit qu'il y a épistasie lorsqu'un ou plusieurs gènes masquent ou empêchent l'expression de facteurs situés à d'autres lieux génétiques. Des études ont montré que les méthodes d'optimisation conventionnelles étaient performantes pour les problèmes quasi linéaires, c'est-à-dire avec une épistasie relativement faible, les AGs eux, sont des solutions appropriées pour des problèmes non linéaires et à épistasie moyenne. Dans le cas où l'on a des problèmes hautement épistatiques, l'obtention de la solution est hautement improbable ($\frac{1}{\text{Card}(X)}$, sans oublier le coût algorithmique pour comparer $\forall x_i \in X, f(x_i)$ (un coût qu'on ne doit pas considérer si l'on cherche $f(x_i = x_0) = 0$)) et nous ne disposons pas de méthodes précises. Dans ce cas-ci, la puissance de calcul est un facteur très important dans l'obtention de la solution.

Une autre question importante en relation avec l'épistasie peut être formulée. Est-ce que des parents à hautes fitness produisent forcément des parents à fitness égales ou plus élevées ? Dans la nature, on peut observer que cette propriété est quasi tout le temps respectée et pourtant dans le cas des algorithmes génétiques, le fait que cette propriété soit vérifiée repose en fait sur le choix de l'algorithme d'encodage fait par l'utilisateur.

Essayons de construire un problème d'optimisation qui pourrait mettre en échec l'AG. Le problème d'épistasie ne se pose pas dans le cas où $n = 1$ puisque l'on a le tout qui est égal à "l'unité", i.e, on a que 2 deux schémas possibles. Si l'on se place, cette fois-ci avec $n = 2$, assumons, WLOG, que le maximum global est ici $11 \in \{1, *\} \cap \{*, 1\}$, on souhaite donc, produire des schémas binaires tels que :

$$(f(0*) > f(1*)) \oplus (f(*0) > f(*1)). \tag{4.4}$$

C'est-à-dire que l'on souhaite construire une fonction de fitness f telle qu'au moins un $H \in \{1, *\} \cap \{*, 1\} \wedge H \neq 11$ ait une fitness plus basse qu'un ou plusieurs schémas ne remplissant aucune de ces conditions, i.e (petit exercice de rédaction logique) :

$$\exists((H \in V = \{1, *\} \cap \{*, 1\} \wedge H \neq S) \wedge (\bar{H} \notin V \wedge \bar{H} \neq S))|f(H) < f(\bar{H}),$$

c'est équivalent à dire qu'on a dans notre cas :

$$\frac{f(00) + f(01)}{2} > \frac{f(10) + f(11)}{2} \tag{4.5}$$

$$\oplus$$

$$\frac{f(00) + f(10)}{2} > \frac{f(01) + f(11)}{2}. \tag{4.6}$$

Les deux relations ne peuvent pas être respectées en même temps sinon cela contredirait l'hypothèse de départ que 11 est notre valeur optimale. WLOG, on choisit la première condition pour la suite.

Dans le but de simplification, on normalise les différentes valeurs de fitness par rapport au complément de notre maximum (donc ici $A \oplus 1 = \bar{A}, A = 11, \bar{A} = 00$).

Posons donc :

$$r = \frac{f(11)}{f(00)}, \qquad c = \frac{f(01)}{f(00)}, \qquad c_2 = \frac{f(10)}{f(00)}.$$

Notre condition de maximum implique les relations suivantes :

$$r > 1 = \frac{f(00)}{f(00)}, \qquad r > c, \qquad r > c_2.$$

On en déduit d'après (4.4) et (4.6) :

$$r < 1 + c - c_2,$$
$$c_2 < 1,\ c_2 < c.$$

Ainsi, on a deux types possibles de fonction f trompeuses pour des strings de longueur $n = 2$ (disons 2-bits) basées sur (4.4).

$$\text{TypeI} : \qquad f(01) > f(00)\ (c > 1).$$
$$\text{TypeII} : \qquad f(01) \leq f(00)\ (c \leq 1).$$

La figure 5 donne une représentation visuelle des deux types de problèmes d'encodage trompeurs. Ces deux fonctions de fitness ne sont pas linéaires. En ce sens, l'épistase est la propriété derrière le caractère trompeur de ces types.

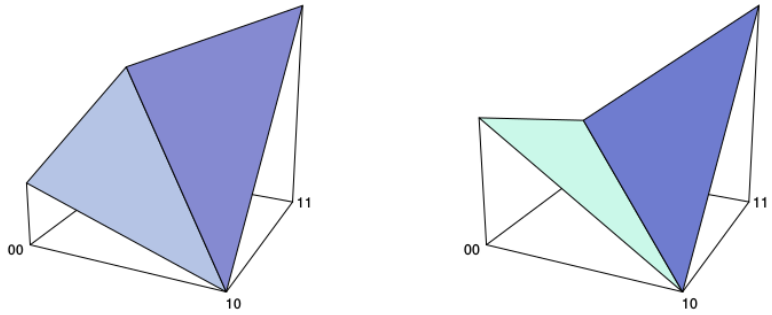


Figure 5: Problème d'encodage de type I (à gauche) et du type II (à droit)

4.2.2 Exemple : problème du voyageur de commerce.

On a déjà mentionné qu'il est essentiel pour un AG que de bons individus produisent un individu comparable voire meilleur. On étudie ici, un exemple de respect de cette caractéristique, appliqué à un problème bien connu des problèmes d'optimisation *problème du voyageur de commerce* (TSP) [14]. On a un ensemble fini de sommets / villes $\{v_1, ..., v_N\}$. Pour chaque pair de villes (v_i, v_j), la distance $D_{i,j}$ est connue. On souhaite trouver une permutation $(p_1, ..., p_n)$ telle que le chemin de v_1 à v_n en passant par les villes intermédiaires, soit le plus court, autrement dit, que la somme de ces distances soit minimale :

$$f(p_{\min}) = \min_{(p_1,\ldots,p_n)} \left\{ \sum_{i=1}^{N-1} D_{p_i,p_{i+1}} + D_{p_n,p_1} \right\}.$$

Ce problème apparaît notamment dans l'optimisation de trajet en électronique pour la confection de circuits imprimés larges.

Afin de résoudre le TSP à l'aide d'un AG, on a besoin d'une méthode d'encodage, d'une méthode de croisement et enfin d'une méthode de mutation. Ces trois routines doivent fonctionner de manière harmonieuse entre elles afin que l'hypothèse des blocs constructifs soit respectée. Il est intelligent d'encoder une permutation comme une string de nombres où l'entrée i correspond au numéro de i-ème ville visitée. Chaque nombre entier entre 1 entre N apparaît qu'une fois afin que le tour soit complet. La méthode de croisement à un point est celle qui paraît la plus appropriée puisqu'ici, on manipule de simples permutations. Des solutions d'encodage sont fournies par [15].

Croisement à map partielle (PMX) Ce croisement a pour but de conserver beaucoup de positions relatives lors des générations. On prend deux strings, on choisit un sous-intervalle de position de ces strings et on les échange, puisque chaque valeur ne peut apparaître qu'une et une seule fois, il y aura des positions de conflits, les valeurs à ces positions seront remplacées par leur équivalent dans ces deux strings. Un exemple est bien plus parlant :

$$p_1 = (1\,2\,3\,4\,5\,6\,7\,8\,9)$$
$$p_2 = (4\,5\,2\,1\,8\,7\,6\,9\,3)$$

On décide que ce sont les positions de 4 à 7 qui sont sélectionnées pour être échangées. Les deux enfants qui en découlent en négligeant les positions de conflits deviennent donc :

$$o_1 = (*\,2\,3\,|\,1\,8\,7\,6\,|\,*\,9)$$
$$o_2 = (*\,*\,2\,|\,4\,5\,6\,7\,|\,9\,3)$$

On remplit les positions de conflit par les valeurs qui ont été échangées dans l'autre enfant à cette même position. Dans notre cas, le 1 et le 4 ont été échangés ainsi que le 8 et le 5.

$$o_1 = (4\,2\,3\,1\,8\,7\,6\,5\,9)$$
$$o_2 = (1\,8\,2\,4\,5\,6\,7\,9\,3)$$

Croisement ordonné (OX) Le croisement ordonné se base sur l'idée que l'ordre des villes a plus d'importance que leur position absolue dans les strings. Comme dans le cas du PMX, OX échange deux sous strings alignées. Les sous strings restantes sont calculées d'une manière différente. Considérons les exemples (p_1, p_2). Échangeons les deux sous strings (toujours de 4 à 7) :

$$o_1 = (*\,*\,*\,|\,1\,8\,7\,6\,|\,*\,*)$$
$$o_2 = (*\,*\,*\,|\,4\,5\,6\,7\,|\,*\,*)$$

Afin de calculer les valeurs aux positions restantes, écrivons les valeurs de p_1, mais en commençant à la position suivant la sous-string d'échange :

$$9\,3\,4\,5\,2\,1\,8\,7\,6$$

Si l'on retire maintenant les valeurs déjà présentes dans l'opération d'échange (de 4 à 7), $(4, 5, 6$ et $7)$:

$$9\,3\,2\,1\,8$$

Maintenant on insère cette liste dans o_2 toujours en partant de la position suivant la sous string d'échange, on a :

$$o_2 = (2\,1\,8\,4\,5\,6\,7\,9\,3).$$

En appliquant cette technique à o_1 :

$$o_1 = (3\,4\,5\,1\,8\,7\,6\,9\,2).$$

Croisement cyclique (CX) PMX et OX ont en commun le fait qu'ils introduisent des allèles en dehors des sites de croisement (d'échange) qui ne sont présents dans aucun des parents à une position choisie. Par exemple (OX), le 3 à la première position dans o_1 ou encore le 8 à la troisième position dans o_2. Le croisement cyclique a pour but de faire face à ce problème, en garantissant le fait que les valeurs à chaque position de n'importe quelle sortie de ce croisement, viennent d'un des deux parents. (p_1, p_2) :

$$p_1 = (1\,2\,3\,4\,5\,6\,7\,8\,9)$$
$$p_2 = (4\,1\,2\,8\,7\,6\,9\,3\,5)$$

On commence à la première position de o_1 :

$$o_1 = (1\ *\,*\,*\,*\,*\,*\,*\,*)$$
$$o_2 = (*\,*\,*\,*\,*\,*\,*\,*\,*)$$

On a donc o_2 qui ne peut avoir que 4 en première position puisqu'on ne souhaite pas de nouvelles valeurs ($\neq 1$ ou $\neq 4$) à cette position :

$$o_1 = (1\ *\,*\,*\,*\,*\,*\,*\,*)$$
$$o_2 = (4\ *\,*\,*\,*\,*\,*\,*\,*)$$

On a donc le 4 qui est fixé pour o_2, il l'est aussi pour o_1 puisqu'on ne peut l'avoir qu'une fois dans o_2, et de même la valeur, à la position du 4 dans o_1, dans o_2 est fixée : 8.

$$o_1 = (1\ **4\ *****)$$
$$o_2 = (4\ **8\ *****)$$

On répète ce processus jusqu'à ce qu'on se retrouve sur une valeur qui a déjà été considérée, i.e on a complété un cycle :

$$o_1 = (1\,2\,3\,4\ ***\ 8*)$$
$$o_2 = (4\,1\,2\,8\ ***\ 3*)$$

Pour le second cycle, on peut prendre comme valeur de départ une valeur de p_2 et l'insérer dans o_1 :

$$o_1 = (1\,2\,3\,4\,7\ **8*)$$
$$o_2 = (4\,1\,2\,8\,5\ **3*)$$

En complétant ce cycle, on obtient :

$$o_1 = (1\,2\,3\,4\,7\ *\ 9\,8\,5)$$
$$o_2 = (4\,1\,2\,8\,5\ *\ 7\,3\,9)$$

Le dernier cycle est ainsi la seule valeur qui n'est pas présente dans o_1 et o_2, (6-6) :

$$o_1 = (1\,2\,3\,4\,7\,6\,9\,8\,5)$$
$$o_2 = (4\,1\,2\,8\,5\,6\,7\,3\,9)$$

Dans le cas où les deux parents sont tels que le croisement s'opère sur un seul cycle, alors aucun croisement n'est effectué ($p_1 \to o_2$ et $p_2 \to o_1$). Des études empiriques ont montré que OX donne en moyenne des résultats 11% et PMX 15% meilleurs que CX. En général, les performances de ces trois méthodes sont pauvres.

Un encodage avec une liste de référence Une différente approche qui modifie l'encodage des strings et telle que toutes les méthodes conventionnelles de croisement sont applicables est celle discutée dans ce paragrahe. Une liste de référence est initialisée : $\{1, ..., N\}$. En commençant sur la première position, on prend l'index de l'élément actuel et on le retire de la liste. Un exemple :

$$p = (1\,2\,4\,3\,8\,5\,9\,6\,7).$$

Le premier élément est 1 et sa position dans la liste de référence est 1. Donc,

$$\tilde{p} = (1\ ********).$$

L'entrée suivante est 2 et sa position dans le reste de la liste $\{2, ..., 9\}$ est 1, donc :

$$\tilde{p} = (1\,1\ *******).$$

La valeur de la position 3 est 4 et sa position dans la liste de référence $\{3, ..., 9\}$ est 2 :

$$\tilde{p} = (1\,1\,2\ ******).$$

En continuant comme cela et en considérant que lorsqu'on est en dehors de la liste, on est à 1 (rem1) on arrive à la conclusion que :

$$\tilde{p} = (1\,1\,2\,1\,4\,1\,3\,1\,1).$$

La remarque 1 qui est précisée laisse supposer qu'une string avec cet encodage est une permutation valide, si et seulement si l'inégalité suivante est vérifiée pour tout $1 \leq i \leq N$:

$$1 \leq \tilde{p}_i \leq N - i + 1,$$

i.e $\tilde{p}_i$ est dans la liste de référence restante au moment i.

Ce critère ne s'applique que sur chaque valeur de la string, de manière complètement indépendante des autres valeurs aux différentes positions, il ne peut donc être violé par aucune méthode de croisement abordée, chaque permutation est donc valide. L'exemple suivant, montre, tout de même, qu'un croisement à un point sur ce type d'encodage, produit des valeurs plus ou moins aléatoires en dehors du site de croisement lorsque l'on calcule o_1 et o_2.

$$\tilde{p}_1 = (1\,1\,2\,1|4\,1\,3\,1\,1)\ p_1 = (1\,2\,4\,3\,8\,5\,9\,6\,7)$$
$$\tilde{p}_2 = (5\,1\,5\,5|5\,3\,3\,2\,1)\ p_2 = (5\,1\,7\,8\,9\,6\,4\,3\,2)$$

$$\tilde{o}_1 = (1\,1\,2\,1|5\,3\,3\,2\,1)\ o_1 = (1\,2\,4\,3\,9\,8\,7\,6\,5)$$
$$\tilde{o}_2 = (5\,1\,5\,5|4\,1\,3\,1\,1)\ o_2 = (5\,1\,7\,8\,6\,2\,9\,4\,3)$$

Le lien entre $(\tilde{p}_1, \tilde{p}_2)$ et $(\tilde{o}_1, \tilde{o}_2)$ est visible, le fait que l'on ait choisi la méthode de croisement à un point se devine, ce n'est absolument pas le cas pour $(p_1, p_2) \rightarrow (o_1, o_2)$ et il est donc sûr d'affirmer qu'il existe des couples pour lesquels la méthode du croisement à un point n'a pas autant d'intérêt qu'avec un supposé autre encodage (l'exemple ci-dessus en est déjà un cas).

Opérateur de recombinaison de bordures (ERO) Les positions des valeurs dans la string n'ont que peu d'importance lorsque prises de manière isolée, leurs ordres relatifs donnent déjà plus d'informations sur la qualité de la permutation mais celle-ci est véritablement définie par l'ensemble des liaisons qu'ont les villes entre elles. En comparaison, les blocs constructifs du problème TSP sont cachés dans ces connexions. La méthode de recombinaison (ERO) exploite cette idée en créant des enfants à partir de la liste des voisins qu'a chaque ville. Prenons comme exemple :

$$p_1 = (1\,2\,3\,4\,5\,6\,7\,8\,9)$$
$$p_2 = (4\,1\,2\,8\,7\,6\,9\,3\,5)$$

Ecrivons l'union de la liste décrite pour p_1 et pour p_2 ($\{2,4,9\} \cup \{2,4\} = \{2,4,9\}, ...$) :

$$1 \rightarrow \{2,4,9\}$$
$$2 \rightarrow \{1,3,8\}$$
$$3 \rightarrow \{2,4,5,9\}$$
$$4 \rightarrow \{1,3,5\}$$
$$5 \rightarrow \{3,4,6\}$$
$$6 \rightarrow \{5,7,9\}$$
$$7 \rightarrow \{6,8\}$$
$$8 \rightarrow (\{2,7,9\}$$
$$9 \rightarrow \{1,3,6,8\}$$

Pour appliquer l'opérateur, on commence par choisir la ville qui a le plus petit nombre de voisins (7 dans notre exemple), on l'inscrit à la première position dans notre enfant et on efface la ville choisie des listes. On s'intéresse ensuite aux voisins de la ville choisie (7 : 6 et 8) : pour la deuxième position, on assigne la ville voisine à celle choisie précédemment, qui a le moins de voisins. Si on se trouve dans le cas où les nombres de voisins des voisins de la ville originalement choisie sont égaux alors, on laisse place à un choix aléatoire.

On répète cette opération jusqu'à ce que la permutation soit complète ou que l'on est face à un conflit (plus de voisins disponibles et pourtant, la permutation est incomplète). Des études statistiques ont montré que la probabilité de ne pas faire face à un conflit est de 98% [16], une probabilité assez importante pour qu'après un second lancement, on ne tombe plus face à un conflit. En suivant cette méthode avec notre exemple, on trouve :

$$o = (7\,6\,5\,4\,1\,9\,8\,2\,3).$$

Il existe bien évidemment plusieurs variantes pour améliorer la convergence d'un AG avec ERO (mais aussi pour CX, OX et PMX).

5 Programmation génétique

> "How can computers learn to solve problems without being explicitly programmed? In other words, how can computers be made to do what is needed to be done, without being told explicitly how to do it?" [17]
>
> — John R. Koza

Les mathématiciens et les chercheurs en informatique, dans leur quotidien professionnel, ne font rien d'autre que de trouver des programmes permettant de résoudre des problèmes donnés. Ils essayent de créer de tels programmes en se basant sur leurs connaissances du problème, sur les principes régissant le problème, sur leurs connaissances des mathématiques et de l'informatique et enfin sur leur intuition. On serait tenté de répondre à la question de John R. Koza, que l'on est simplement incapable de créer de tels programmes et pourtant, pour les développer, on peut simplement appliquer les principes des algorithmes génétiques, en faisant des modifications appropriées aux opérateurs des AGs. C'est-à-dire, implémenter des méthodes de croisement et de mutation favorables à la résolution d'un certain problème. Les opérateurs de sélection et d'échantillonnage, eux, n'ont pas nécessairement besoin d'être modifiés puisque ces méthodes ne tiennent pas compte de la représentation du problème.

Est-ce que la programmation génétique peut fonctionner ? Koza dans [17], propose une hypothèse.

Hypothèse de la programmation génétique. *Sont fournis, un problème résoluble, une définition d'un langage approprié de programmation et un ensemble assez grand d'exemples représentatifs (c'est-à-dire des paires (entrée, sortie) correctes), un algorithme génétique est normalement capable de trouver un programme qui (approximativement au moins) résout le problème.*

On peut penser que cette hypothèse n'est qu'une histoire de croyance puisque personne n'a encore réussi à la prouver (si elle est prouvable). Au lieu de fournir une preuve, Koza propose un grand ensemble d'exemples qui appuient cette hypothèse. Les problèmes suivants ont correctement été résolus par de la programmation génétique :

- Processus de contrôle : Pendule inversé (l'algorithme génétique doit faire en sorte que le pendule soit maintenu dans une position d'équilibre instable à 180°).
- Logistique : contrôle simple de robots, problème d'empillement (empiler N blocs rectangulaires rigides et identiques de manière stable sur une table et cela en maximisant le contact entre les blocs et le vide).
- Générateur de nombres pseudo-aléatoires, création d'ANNs.
- Stratégique : Poker, Tic Tac Toe, démineur, Super Mario Bross [18] et [19].
- Cinématique inverse (calcul des positions et des rotations d'un modèle afin d'obtenir une pose désirée).

- Classification
- Calcul symbolique (formel) :
  - Découverte de formules de récurrence (séquence de Fibonacci, etc).
  - Approximation d'un ensemble de points par une formule littérale.
  - Résolution d'équations (dont des équations diophantines, la résolution n'étant absolument pas systématique).
  - Calcul différentiel et intégral (découverte d'identités).
  - Découverte de différentes identités mathématiques, notamment trigo-nométriques (les expressions de définition des fonctions trigonométri-ques n'étant pas fournies à l'AG, il doit se contenter de manipuler des fonctions ne présentant pas de règle algébrique particulière au premier abord).

Ce chapitre a pour but de donner une introduction à la programmation génétique en fournissant des exemples basiques d'applications de ceux-ci.

5.1 Représentations possibles d'un programme génétique

On peut premièrement penser à considérer un programme comme une chaîne de caractères, deux grandes limites de cette méthode de représentation font que c'est impossible de penser un programme d'une telle manière dans le cadre des AGs :

1. Il est inapproprié d'assumer que les différents programmes à travers les générations seront de longueur fixe.
2. La probabilité d'obtenir un programme syntaxiquement correct dans un langage de programmation donné, simplement à travers l'initialisation, le croisement et la mutation, est extrêmement faible (quasi nulle).

On doit donc trouver un moyen de représenter le programme tel que la syntaxe de celui-ci est facilement garantie comme correcte. L'approche la plus commune pour représenter un programme dans le cadre des AGs est de considérer ces programmes comme des arbres. Ainsi, l'initialisation est la génération de l'arbre, le croisement est effectué en échangeant des sous-arbres et le remplacement aléatoire de certains sous-arbres peut servir comme opération de mutation.

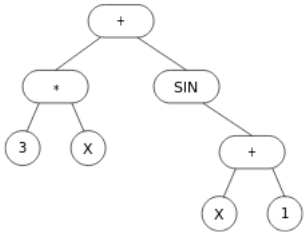


Figure 6: Représentation en arbre de (+(*3X)(SIN(+X 1)))

Cette structure est une liste imbriquée, les programmes écrits dans des langages tels que LISP ont ces types de structures en arbre. La figure 6 montre un exemple de représentation de la fonction $3x + \sin(x + 1)$ dans un langage tel que LISP. De cette représentation, on devine que les constantes et les variables sont les feuilles de l'arbre alors que les fonctions sont les branches de l'arbre. Cette représentation est rapidement limitée lorsque l'on souhaite exprimer des expressions logiques simultanément à des chaînes de caractères ou à des données numériques.

Une approche plus générale qui nous permet d'avoir le plus de flexibilité dans la représentation d'un programme utilise la représentation en arbre des sous-programmes dans leur forme de Backus-Naur. Là encore, des exemples sont plus parlants, considérons le langage suivant qui opère seulement sur des valeurs binaires.

$$
\begin{aligned}
\text{S} &:= \langle \text{exp} \rangle; \\
\langle \text{exp} \rangle &:= \langle \text{var} \rangle | \text{"("} \langle \text{neg} \rangle \langle \text{exp} \rangle \text{")"} | \text{"("} \langle \text{exp} \rangle \langle \text{bin} \rangle \langle \text{exp} \rangle \text{")"}; \\
\langle \text{var} \rangle &:= \text{"}x\text{"} | \text{"}y\text{"}; \\
\langle \text{neg} \rangle &:= \text{"NON"}; \\
\langle \text{bin} \rangle &:= \text{"ET"} | \text{"OU"};
\end{aligned}
$$

Pour rapidement introduire la représentation BNF, les symboles dans les crochets angulaires $\langle\rangle$ sont ce qu'on appelle des symboles non-terminaux, i.e. des symboles que l'on doit développer en se référant à leur définition. Les symboles entre guillemets sont eux appelés symboles terminaux, i.e. on ne peut plus les développer. La première règle du langage : S:=$\langle$exp$\rangle$; est appelée symbole de lancement. Une règle BNF sous la forme :

$$\langle \text{regle} \rangle := \langle \text{deriv}_1 \rangle | \langle \text{deriv}_2 \rangle | ... | \langle \text{deriv}_n \rangle;$$

définit comment ce symbol non-terminal peut être développé, les différentes possibilités étant séparées par des barres verticales. Essayons d'écrire l'expression logique (NON (x OU y)) à l'aide de la représentation BNF.

1. Symbole de départ : S := $\langle$exp$\rangle$
2. On remplace $\langle$exp$\rangle$ par sa seconde dérivée possible dans notre langage :

$$\langle \text{exp} \rangle \longrightarrow (\langle \text{neg} \rangle \langle \text{exp} \rangle)$$

3. Le symbole $\langle$neg$\rangle$ ne peut qu'être développé en un symbole terminal NON :

$$(\langle \text{neg} \rangle \langle \text{exp} \rangle) \longrightarrow (\text{NON} \langle \text{exp} \rangle)$$

4. On remplace $\langle$exp$\rangle$ par sa troisième dérivation possible :

$$(\text{NON} \langle \text{exp} \rangle) \longrightarrow (\text{NON}(\langle \text{exp} \rangle \langle \text{bin} \rangle \langle \text{exp} \rangle))$$

5. On développe la seconde dérivation possible pour $\langle$bin$\rangle$:

$$(\text{NON}(\langle \text{exp} \rangle \langle \text{bin} \rangle \langle \text{exp} \rangle)) \longrightarrow (\text{NON}(\langle \text{exp} \rangle \text{OU} \langle \text{exp} \rangle))$$

6. On remplace les deux occurences de $\langle$exp$\rangle$ par leur première dérivation, $\langle$var$\rangle$:

$$(\text{NON}(\langle \text{exp} \rangle \text{OU} \langle \text{exp} \rangle)) \longrightarrow (\text{NON}(\langle \text{var} \rangle \text{OU} \langle \text{var} \rangle))$$

7. Enfin, on remplace les deux occurences de $\langle$var$\rangle$ par x et y respectivement :

$$(\text{NON}(\langle \text{var} \rangle \text{OU} \langle \text{var} \rangle)) \longrightarrow (\text{NON}(x \text{OU} y))$$

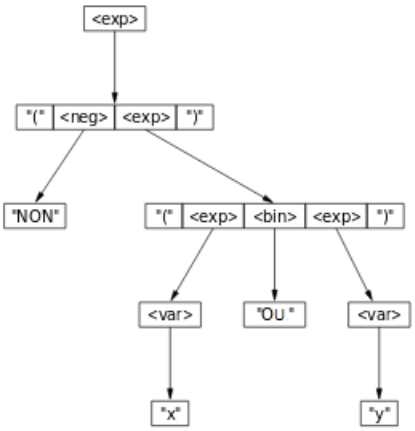


Figure 7: L'arbre de dérivation BNF de (NON(x OU y))

5.1.1 Choix du langage

L'hypothèse de Koza affirme que le langage de programmation doit être choisi de manière à ce que le problème puisse être résolu. Cela n'implique pas forcément que l'on doit choisir un langage tel que tout un ensemble large de problèmes puissent être résolus. Il est évident que l'espace de recherche augmente avec la complexité du langage. On sait que la taille de l'espace de recherche influence les performances de l'AG, plus il est large, plus l'AG sera lent à trouver la solution (en assumant que celui-ci permet de résoudre le problème). Il est donc recommandé de restreindre le langage aux constructions nécessaires en évitant les expressions superflues. Par exemple, si l'on cherche à faire une régression symbolique (approcher un ensemble de points par une formule littérale) mais que nous sommes intéressés que par des polynômes à coefficients entiers, il serait inutile et non optimisé d'introduire dans notre langage des coefficients rationnels (et non entiers) ou des fonctions exponentielles. Un bon choix serait :

$$
\begin{aligned}
\text{S} &:= \langle \text{fonc} \rangle; \\
\langle \text{fonc} \rangle &:= \langle \text{var} \rangle | \langle \text{const} \rangle | \text{"("} \langle \text{fonc} \rangle \langle \text{bin} \rangle \langle \text{fonc} \rangle \text{")"}; \\
\langle \text{var} \rangle &:= \text{"}x\text{"}; \\
\langle \text{const} \rangle &:= \langle \text{int} \rangle | \langle \text{const} \rangle \langle \text{int} \rangle; \\
\langle \text{int} \rangle &:= \text{"0"} | ... | \text{"9"}; \\
\langle \text{bin} \rangle &:= \text{" + "} | \text{" − "} | \text{" ∗ "};
\end{aligned}
$$

Pour représenter une fonction à coefficients rationnels mais en utilisant seulement des integers, il suffit de rajouter le symbole de la division "/" aux dérivations de l'opérateur $\langle \text{bin} \rangle$.

Un autre exemple approprié pour découvrir des identités trigonométriques :

$$
\begin{aligned}
\text{S} &:= \langle \text{fonc} \rangle; \\
\langle \text{fonc} \rangle &:= \langle \text{var} \rangle | \langle \text{const} \rangle | \langle \text{trig} \rangle \text{"("} \langle \text{fonc} \rangle \text{")"} | \text{"("} \langle \text{fonc} \rangle \langle \text{bin} \rangle \langle \text{fonc} \rangle \text{")"}; \\
\langle \text{var} \rangle &:= \text{"}x\text{"}; \\
\langle \text{const} \rangle &:= \text{"0"} | \text{"1"} | \text{"}\pi\text{"}; \\
\langle \text{trig} \rangle &:= \text{" sin "} | \text{" cos "}; \\
\langle \text{bin} \rangle &:= \text{" + "} | \text{" − "} | \text{" ∗ "};
\end{aligned}
$$

5.2 Manipulation des programmes

On a une méthode d'encodage de nos programmes, les arbres BNF. Il nous reste plus qu'à définir les trois opérations suivantes : initialisation aléatoire, croisement et mutation.

5.2.1 Intialisation aléatoire

Jusque-là, on n'a pas prêté attention à la création de la population initiale. On a émis l'hypothèse que l'on pourrait simplement en générer une aléatoirement avec une certaine distribution de probabilités (souvent uniforme). Cette initialisation est très simple à implémenter dans le cas de strings binaires à longueur fixée, la génération aléatoire d'arbres BNF semble déjà plus complexe à effectuer.

Il y a deux variantes basiques pour générer des programmes aléatoires respectant un certain langage BNF :

1. En débutant au symbole S, il est possible de développer des symboles non-terminaux de manière récursive, où l'on choisit aléatoirement les dérivations si on en a plusieurs possibles. Cette approche est simple et rapide mais présente plusieurs désavantages : Premièrement, il est quasi impossible de réaliser une distribution uniforme des expressions. Deuxièmement, il est important d'établir des contraintes notamment sur la taille de l'arbre autrement dit la profondeur des développements et cela pour éviter l'accroi-ssement excessif des programmes. La difficulté de cette tâche dépend forcément de la complexité du langage BNF choisi, en effet, dans certains cas, il peut être compliqué de ralentir puis d'arrêter l'accroissement du programme.

2. Une autre méthode peut être de lister tous les arbres possibles d'une certaine profondeur et ensuite en sélectionner un aléatoirement. Les problèmes de profondeur et de sélection uniforme sont résolus par cette méthode mais elle présente un gros défaut, en effet le calcul de tous ces arbres est coûteux.

5.2.2 Croisement de programmes

Si notre programme est dans la forme BNF, une méthode de croisement simple pourrait être l'échange de sous-arbres sélectionnés. Dans le cas où les sous-arbres présentent différents types de données, c'est-à-dire qu'ils représentent, par exemple, des expressions logiques et des expressions numériques, le croisement n'est pas forcément correct, les expressions croisées doivent être donc de classes compatibles dans la syntaxe. Les arbres en dérivation ne posent pas ce problème puisque seules les expressions compatibles sont reliées entre elles dans la définition du langage.

Prenons les exemples suivants :

$$(\mathrm{NON}(x\mathrm{OU}y)),$$

$$((\mathrm{NON}x)\mathrm{OU}(x\mathrm{ET}y)).$$

La figure 8, montre comment ces deux autres arbres

$$(\mathrm{NON}(x\mathrm{OU}(x\mathrm{ET}y))),$$

$$(\mathrm{NON}(x\mathrm{OU}y))$$

sont obtenus.

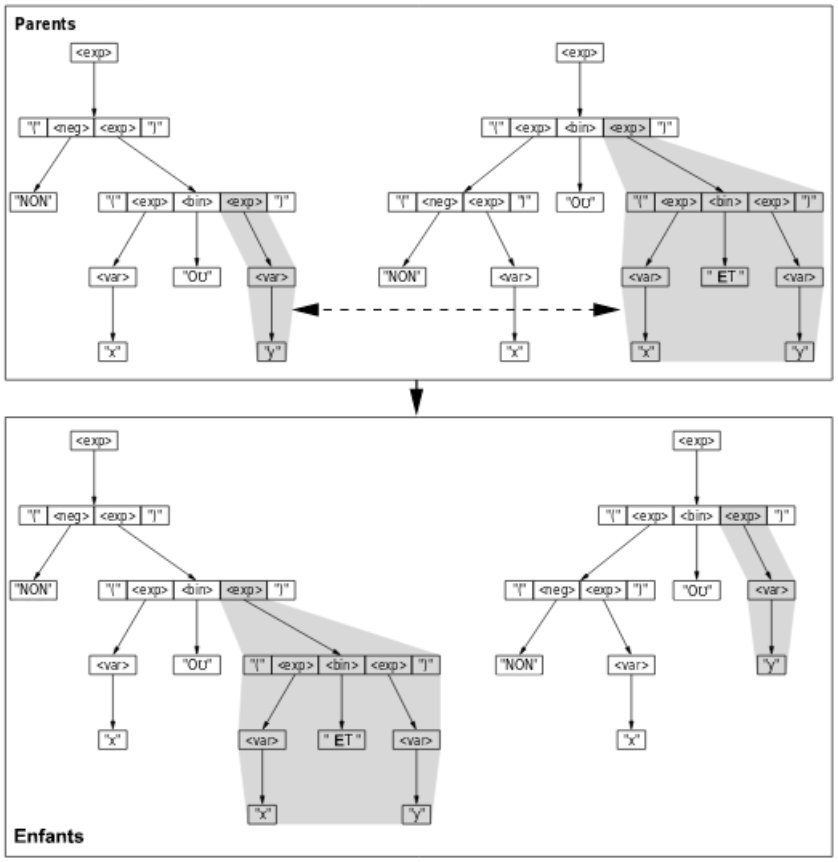


Figure 8: Croisement des deux arbres de dérivations

5.2.3 Méthode de mutation de programmes

La mutation jusque-là, est la déformation d'une petite partie d'un chromosome. L'opération de mutation, la plus commune aux arbres est donc le remplacement d'un sous-arbre sélectionné, le remplacement peut être effectué des deux manières d'initialisation de la population présentées dans (5.2.1), bien sûr, on ne commence pas forcément au symbole S mais à l'expression non-terminale, la moins profonde du sous-arbre considéré. La figure 9, montre un exemple où, dans l'expression logique (NON(xOUy)), la variable y est remplacée par NON(y).

5.2.4 Fonction de fitness

La fonction de fitness doit être choisie en fonction du problème et il n'existe donc pas de recette particulière (si ce n'est certaines conditions, la fonction de fitness doit nous renseigner sur la qualité de chaque individu) pour la choisir. La plupart du temps, la fonction de fitness est définie par une certaine différence entre la sortie souhaitée et la sortie obtenue. Koza par exemple, utilise la somme des erreurs quadratiques (chaque terme est donc positifs) entre les images souhaitées et les images obtenues par la fonction F, fonction représentative de notre programme.

$$f(F) = \sum_{i=1}^{N} (y_i - F(x_i))^2.$$

La liste $(x_i, y_i)_{i \leq i \leq N}$ est la liste de référence de départ, celle fournie par l'utilisateur, chaque paire étant composée d'une sortie y_i souhaitée associée à x_i.

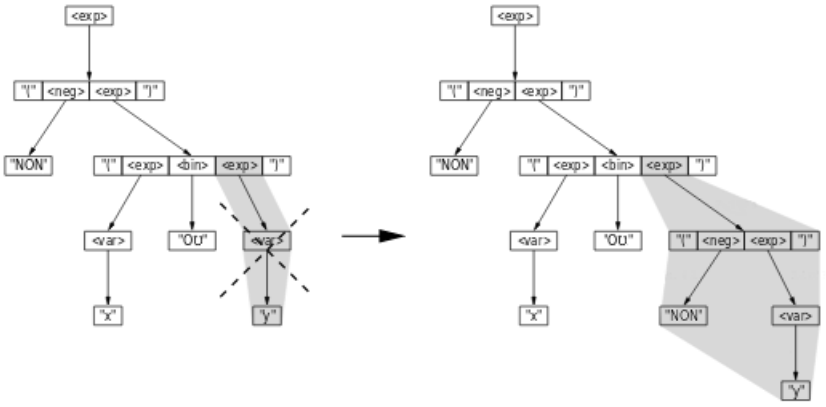


Figure 9: Croisement des deux arbres de dérivations

Les exemples doivent être choisis tels qu'ils décrivent bien le problème d'origine en couvrant correctement l'espace de celui-ci. Dans la plupart des cas, on cherche à minimiser les fonctions de fitness numériques et dans notre cas $f(F)$ doit être minimisée. Pour obtenir une fonction de fitness qui doit être forcément minimisée à partir d'une première fonction de fitness, on peut appliquer par exemple ces transformations :

$$f_m(b_{i,t}) = f(b_{i,t})) - \max_{b_{j,t} \in \mathcal{P}_t} f(b_{j,t}),$$

dans le cas où f doit être maximisée, ou

$$f_m(b_{i,t}) = f(b_{i,t})) - \min_{b_{j,t} \in \mathcal{P}_t} f(b_{j,t}),$$

lorsque f doit être minimisée. Si l'on veut une fonction de fitness à maximiser, on peut appliquer la transformée suivante :

$$f_M(b_{i,t}) = \max_{b_{j,t} \in \mathcal{P}_t} f_m(b_{j,t}) - f_m(b_{i,t}).$$

5.3 Récapitulation

En conclusion, pour l'application d'un algorithme génétique sur un problème donné, il nous faut :

1. Une fonction de fitness appropriée qui fournit assez d'information pour guider l'AG à la solution.

2. Une description syntaxique d'un langage qui contient les éléments nécessaires pour résoudre le problème.

3. Un interpréteur du langage.

6 Démonstrations d'AG

On ne peut terminer ce TPE sans fournir des applications concrètes des AGs. Le premier exemple que l'on va aborder, est un algorithme génétique GMaths, écrit par Yvann Le Fay qui à un ensemble de points dans $\mathbb{R}^{n\in\mathbb{N}}$ associe une expression littérale passant par ces points. Le second exemple est plus pédagogique, voire amusant mais les applications sont moindres, EvoLisa, un AG codé par Roger Johansson [20].

6.1 GMaths

GenMath est un algorithme génétique écrit en C# [.NET] qui a pour but d'interpoler un ensemble de points. Pour réutiliser les termes théoriques présentés au cours de ce TPE, l'espace de recherche est composé des différentes constantes, fonctions et variables pouvant être utilisées pour arriver à une certaine solution. En entrée, on lui fournit donc cet espace de recherche (le langage sous la forme BNF) et bien sûr, l'ensemble de points à interpoler, en sortie, le programme doit nous renvoyer une ou plusieurs fonctions (pouvant être équivalentes, le calcul formel n'étant pas implémenté dans l'algorithme, il ne peut pas simplifier, développer et factoriser de manière systématique) passant par les points d'entrées. On se doute bien, qu'il y a énormément de cas dans lesquels cet algorithme peut être utile, notamment si l'on souhaite approcher un phénomène physique mathématiquement à partir de mesures.

L'initialisation de la population de départ dans un langage donné est effectuée de la première manière présentée dans (5.2.1), l'opérateur de croisement est celui représenté dans la Figure 8, la mutation est le simple remplacement d'un sous-arbre et enfin la fonction de fitness est $\sum_{i=1}^{N} |(y_i - F(x_{1,i}, ..., x_{n,i}))|$. On peut dorénavant donner des cas pratiques.

6.1.1 Exemples

La fonction recherchée est $F_r(x, y) = x\sqrt{\frac{x}{y}}^{y}$ et l'ensemble de points choisis tel que l'ensemble des $F_r(x, y)$ soient relativement petits et surtout entiers pour faciliter l'emploi de l'algorithme. En effet, il nous faut fournir que des valeurs numériques "simples", sans expression telle qu'une racine carré, la précision des variables "Double" en .NET n'est que de 15 à 16 chiffres, les entrées seraient lourdes.

x	y	$F_r(x, y)$
64	4	16384
9	1	27
26	2	338
36	6	7776

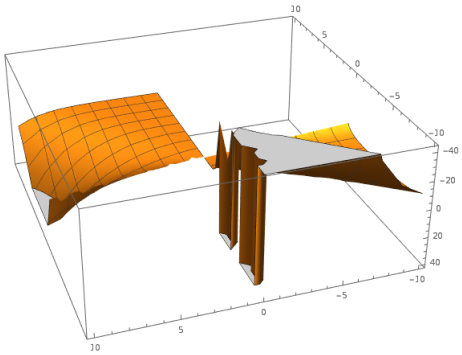


Figure 10: Plot3D[$x\sqrt{\frac{x}{y}}^{y}$, {x, -10, 10}, {y, -10, 10}, ImageSize $\longrightarrow$ Large], modélisation de F_r en utilisant Mathematica

On a préféré ne fournir que peu d'exemples puisque plus il y a d'exemples, plus le temps de calcul est important même si on augmente la probabilité d'obtenir une fonction qui est la fonction recherchée. Celle-ci étant relativement courte, il y a de grandes chances pour que celle obtenue par l'AG soit celle souhaitée. Le langage utilisé est restreint pour réduire ce temps de calcul bien que l'on puisse retirer $\mathbb{R}$ puisque notre fonction ne fait pas intervenir de coefficient :

$$\begin{aligned} S &:= \langle \text{fonc} \rangle; \\ \langle \text{fonc} \rangle &:= \langle \text{var} \rangle | \langle \text{double} \rangle | \text{"("} \langle \text{fonc} \rangle \langle \text{bin} \rangle \langle \text{fonc} \rangle \text{")"}; \\ \langle \text{var} \rangle &:= \text{"}x\text{"} | \text{"}y\text{"}; \\ \langle \text{double} \rangle &:= \text{"}\mathbb{R}\text{"}; \\ \langle \text{bin} \rangle &:= \text{"}/\text{"} | \text{"} * \text{"} | \text{"} \wedge \text{"} | \text{"}sqrt\text{"}; \end{aligned}$$

On obtient en sortie

```
[hallelujah@pc31 ~]$ mono
    '/run/media/hallelujah/UUI/demonstration_gmath/GenMath_MD/GenMath_MD/bin/Debug/GenMath_MD.exe'
Cas.Length : 4
x_1 = 64 y_1 = 4 z_1 = 16384 x_2 = 9 y_2 = 1 z_2 = 27 x_3 = 26 y_3 = 2 z_3 = 338 x_4 = 36 y_4 = 6 z_4 =
    7776
Err : 24524.0855575878, Gen : 1 -> (4.73864690854151 / x)
Err : 24511.5395001972, Gen : 2 -> (4.10379481460144^0.859448039373126)
Err : 24496.0393483318, Gen : 7 -> (-2.43491510741176 / (-4.23603570518831 / (-4.73277012805118 *
    -2.66139119288949)))
Err : 24467.9961453882, Gen : 10 -> ((sqrt(8.90199619759898)) * 4.77639824607242)
Err : 24463.5179892523, Gen : 11 -> (-2.43491510741176 / (-4.23603570518831 / (-4.73277012805118 *
    -5.65)))
Err : 24408.9745146846, Gen : 13 -> (x * 0.859448039373126)
Err : 24385.9610419505, Gen : 17 -> (8.90199619759898 * 4.77639824607242)
Err : 24317.7791643272, Gen : 20 -> (x * 1.53496915313181)
Err : 23916.537, Gen : 30 -> (4.739 * x)
Err : 23757.3623200247, Gen : 45 -> (x * 6.09946735021633)
Err : 23702.7233249565, Gen : 51 -> (x^(sqrt(2.190427634488806)))
Err : 23176.2623680654, Gen : 52 -> (x * 11.0661336062784)
Err : 21074.2207150232, Gen : 61 -> ((x * 3.78254450335286) * 11.0661336062784)
Err : 21014.9095819953, Gen : 63 -> ((8.90199619759898 * x) * 4.80461020944855)
Err : 18504.5432051678, Gen : 67 -> ((x * 7.35502081334359) * 11.0661336062784)
Err : 11314.3997296868, Gen : 68 -> ((8.90199619759898 * (4.48927854629666 * x)) * 4.80461020944855)
Err : 10021.7477082898, Gen : 71 -> ((11.780984940371 * (4.48927854629666 * x)) * 4.80461020944855)
Err : 9531.16981940468, Gen : 111 -> (x * (-10.404368748611 / (-2 / x)))
Err : 7883.623, Gen : 113 -> ((4.739 * x) * x)
Err : 1068, Gen : 121 -> ((y * x) * x)
Err : 1068, Gen : 587522 -> ((((sqrt(x)) * y) * (sqrt(x))) * x)
Err : 203.786820077143, Gen : 616657 -> (x * ((sqrt(y))^(sqrt(x))))
Err : 203.78682007714, Gen : 623362 -> ((sqrt((y^(sqrt(x))))) * x)
Err : 203.78682007714, Gen : 636276 -> ((sqrt((y^(x / (sqrt(x)))))) * x)
Err : 0, Gen : 639001 -> ((sqrt(((x / y)^y))) * x)
Generations : 639001

((sqrt(((x / y)^y))) * x)

Temps : 00:03:04.8380362

x = 64, y = 4, f(x,y) = 16384, g(x,y) = 16384
x = 9, y = 1, f(x,y) = 27, g(x,y) = 27
x = 26, y = 2, f(x,y) = 338, g(x,y) = 338
x = 36, y = 6, f(x,y) = 7776, g(x,y) = 7776
```

Un second exemple faisant intervenir des fonctions trigonométriques,
$F_r(x, y) = \cos x \sin y + \sin xy$, pour résoudre le problème des valeurs numériques non exactes, les images sont calculées lors de l'exécution du programme.

x	y	$\approx F_r(x, y)$
$1/2\pi$	$3/2\pi$	0.899672165311148
$1/6\pi$	$1/7\pi$	0.608588165699673
$25/26\pi$	$4/9\pi$	−1.85778162284019
π	π	−0.430301217000092

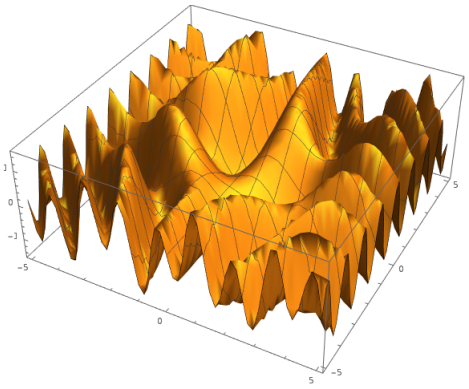


Figure 11: Plot3D[$\cos x \sin y + \sin xy$, {x, -5, 5}, {y, -5, 5}, ImageSize $\longrightarrow$ Large]

$$
\begin{aligned}
\text{S} &:= \langle \text{fonc} \rangle; \\
\langle \text{fonc} \rangle &:= \langle \text{var} \rangle | \langle \text{trig} \rangle \text{"("} \langle \text{fonc} \rangle \text{")"} | \text{"("} \langle \text{fonc} \rangle \langle \text{bin} \rangle \langle \text{fonc} \rangle \text{")"}; \\
\langle \text{var} \rangle &:= \text{"}x\text{"} | \text{"}y\text{"}; \\
\langle \text{trig} \rangle &:= \text{" sin "} | \text{" cos "}; \\
\langle \text{bin} \rangle &:= \text{" + "} | \text{" * "};
\end{aligned}
$$

Sortie (seulement la fin) :

```
...
Err : 0.00856636558281, Gen : 125623 -> (0 + ((sin(((cos(x)) * y))) + (sin((y * x)))))
Err : 0, Gen : 149930 -> (0 + ((sin(((y * x) + 0))) + ((sin(y)) * ((cos(x)) + 0))))
Generations : 149930

(0 + ((sin(((y * x) + 0))) + ((sin(y)) * ((cos(x)) + 0))))

Time : 00:00:48:2037682

...

(0 + ((sin((y * x))) + ((sin(y)) * ((cos(x)) + 0))))
(0 + ((sin((y * x))) + ((sin(y)) * (cos(x)))))
```

Enfin pour le troisième et dernier exemple, on fait intervenir la fonction exponentielle et on se place dans $\mathbb{R}^3$, $F_r(x,y,z) = \frac{x}{y}e^z$.

x	y	z	$\approx F_r(x,y,z)$
4	2	1	5.43656365692
7	5	3	28.1197516925
8	1	3	160.684295386
9	6	6	605.143190239

$$
\begin{aligned}
\text{S} &:= \langle \text{fonc} \rangle; \\
\langle \text{fonc} \rangle &:= \langle \text{var} \rangle | \langle \text{const} \rangle | \text{"("} \langle \text{fonc} \rangle \langle \text{bin} \rangle \langle \text{fonc} \rangle \text{")"}; \\
\langle \text{double} \rangle &:= \text{"}\mathbb{R}\text{"}; \\
\langle \text{var} \rangle &:= \text{"}x\text{"} | \text{"}y\text{"} | \text{"}z\text{"}; \\
\langle \text{const} \rangle &:= \langle \text{double} \rangle | \text{"}e\text{"} | \langle \text{const} \rangle \langle \text{double} \rangle; \\
\langle \text{bin} \rangle &:= \text{" } \wedge \text{ "} | \text{"/"} | \text{" } * \text{ "};
\end{aligned}
$$

Un autre langage qui semble correspondre serait celui-ci sans $\mathbb{R}$ mais après plusieurs tests, il semblerait que l'algorithme bloque à un certain cas,
$F(x,y,z) = e^z$, une des raisons est que sans $\mathbb{R}$, l'AG doit diminuer l'évaluation de la fonction de fitness en ne faisant des mutations qu'avec $\langle \text{var} \rangle$, "$e$" et $\langle \text{bin} \rangle$, or il semblerait que la transition entre e^z et xe^z augmente la fonction de fitness pour les fonctions qu'il génère, et en compétition avec ses parents, l'enfant meurt donc et l'algorithme boucle lorsqu'il mute de cette manière. Dans les cas où l'algorithme mute e^z en ye^z ou ze^z (de même pour les autres arbres possibles), les enfants meurent aussi. Cela est dû au fait, que la mutation privilégie le remplacement d'un sous-arbre en un autre de profondeur incrémentée d'un, autrement dit d'une transition à l'autre, il est difficile d'avoir $e^z \overset{t \to t+1}{\longrightarrow} \frac{x}{y}e^z$ puisque ici il y a en plus de l'utilisation d'un $\langle \text{var} \rangle$, l'utilisation d'un symbole non terminal $\langle \text{bin} \rangle$ et d'un second $\langle \text{var} \rangle$. En rajoutant $\mathbb{R}$ à notre langage, on offre la possibilité à l'algorithme de changer des coefficients pour diminuer la fonction de fitness, à un point critique, les mutations entre ces coefficients et nos variables auront pour rôle de créer un coefficient dépendant d'un symbole $\langle \text{var} \rangle$ et qui respecte le caractère élitiste nécessaire à la prochaine population, la transition aura alors débuté, de nouveau, la mutation d'un coefficient de $\mathbb{R}$ deviendra $\langle \text{bin} \rangle \langle \text{var} \rangle$ et, on aura potentiellement $f(F) = 0$. Pour éviter ce genre de problème, il paraît être une bonne idée que d'effectuer des sélections pas aussi drastiques que de prendre que les deux meilleurs d'une population, ainsi on offre aux individus plusieurs générations pour effectuer leurs mutations potentiellement bénéfiques. On peut aussi encourager les mutations à deux de profondeurs ou plus au lieu d'encourager celles à une seulement. Ici en rajoutant $\mathbb{R}$ et avec un temps de calcul d'environ 14 heures, la fitness la plus faible obtenue sur une des instances du programme était de l'ordre du 9.3×10^{-6}, sans $\mathbb{R}$, de l'ordre du 10^{-2}. Une des instances puisque l'initialisation de la première population est l'un des processus clé dans le temps de calcul potentiel de la solution, il faut partir sur de bonnes bases, ce qui est aléatoire. Même après avoir dépassé la fonction e^z, l'algorithme semble avoir du mal à trouver la bonne fonction et cette fois-ci pour des raisons simples, les fonctions qu'il a dans sa population sont bien trop complexes et la probabilité qu'il mute des arbres aussi longs en des arbres courts comme l'est celui de la fonction recherchée est faible.

6.2 EvoLisa

EvoLisa est un algorithme génétique écrit en C# [.NET] par Roger Johansson [20], qui en utilisant un nombre limité de polygones, approche une image qui est l'entrée du programme. La taille de la population est de 2, un parent et un enfant qui sont en compétition, il n'y a pas de place au croisement ce qui fait de cet algorithme un type bien particulier des algorithmes génétiques, *hill-climbing AGs*. L'image est représentée comme une famille de matrices représentant chaque pixel (code RVB) de l'image $\mathcal{I}$, $C(x, y, \mathcal{I}) = \begin{pmatrix} r \\ v \\ b \end{pmatrix}_{(x,y)}$. La mutation est effectuée par l'altération des polygones dans leur matrice RVB ainsi que par la mutation de la définition même des polygones (voir par l'ajout), c'est-à-dire le nombre de sommets les définissant, ainsi que leur position.

La fonction de fitness utilisée pour comparer $\mathcal{I}_t$ à $\mathcal{I}$ est

$$f(\mathcal{I}_t) = \sum_{C(x,y,\mathcal{I}_t)} (C(x, y, \mathcal{I}_t) - C(x, y, \mathcal{I}))^2.$$

En dehors du côté purement éducatif de cet algorithme, on peut lui trouver des propriétés en compression grossière des données. Un exemple (figure 12).

Figure 12: EvoLisa en action sur Bill Gates [20]

7 Conclusion

Si l'idée même d'un algorithme basé sur les processus d'évolution naturelle ainsi que l'extensibilité des domaines d'application de cet AG, sont tout à fait incroyables, les dérivées de cette classe d'algorithmes en apprentissage machine, c'est-à-dire les réseaux DeepLearning et ConvNet (en excluant les AGs Fuzzy qu'on n'a pas pu aborder), produisent des résultats bien supérieurs en matière d'application, à leurs prédécesseurs. On ne parle plus que de problème d'optimisation au sens simple, mais d'apprentissage avec mémoire, des algorithmes capables d'apprendre un langage rien qu'en lisant des livres écrits dans ce langage, d'autres peignant des tableaux s'inspirant de Van Gogh ou de Léonard de Vinci, créant de toutes pièces, des photos, etc. Des avancées nous rapprochant toujours plus de l'intelligence artificielle forte. Beaucoup d'espoir en ces domaines qui ont, ces vingt dernières années, connu un grand avancement qui s'est effectué à travers notamment les AGs.

A Synthèse

Synthèse de TPE (Yvann Le Fay) : Les algorithmes génétiques.

Lors de leur année de première, les élèves sont amenés à effectuer des travaux personnels encadrés (TPE). Ainsi, pour mon année de première, j'ai travaillé avec Killian Hanger sur les algorithmes génétiques. Ce thème regroupant mes grandes passions : les mathématiques et l'informatique (faisant aussi un parallèle avec la génétique).

Je suis celui qui a proposé le sujet, un sujet que je maîtrise assez bien puisque je m'étais largement renseigné dès la troisième sur ces algorithmes. Nous avons choisi une problématique importante dans la compréhension profonde du fonctionnement de ces algorithmes et présentant de multiples aspects : quels sont les principes mathématiques gouvernant les algorithmes génétiques et leurs applications ?

Le sujet s'est précisé progressivement. En effet, au départ, nous voulions étudier les principes mathématiques derrière les ANNs et les AGs, afin de lever le mystère sur le fonctionnement de classes d'algorithmes permettant d'accomplir des choses tout à fait extraordinaires. Les premiers exemples d'applications que je souhaitais étudier étaient DeepDream (Google), un ANN permettant la reconnaissance de motifs suivant une base de données d'images et le renforcement de ces motifs, en sortie, les images ressemblant à des visions psychédéliques, un ANN permettant le classement de photos selon leur contenu, des OCR et un ANN nommé Neural-Style, un ConvNet recomposant certains styles d'images sur d'autres (par exemple, donner à une entrée graphique, le style artistique de Picasso). Traiter entièrement les principes des ANNs (et CNNs) et AGs ainsi que leurs applications en aussi peu de temps paru bien compliqué. J'ai alors décidé que le sujet de notre TPE se restreindrait aux AGs puisque c'est un domaine que je maîtrise, ayant déjà entièrement écrit un programme en C# [.NET], application basique des fondements des AGs.

Les chercheurs du monde entier utilisent LaTeX, un système de composition de documents, outil extrêmement puissant pour la rédaction de formalisme mathématique, programme complètement open-source. C'est pour cela que le support de ce TPE est un pdf généré par LaTeX. Dans le but de pouvoir travailler en même temps que mon partenaire sur le .tex, j'ai proposé d'utiliser ShareLaTeX, un site d'édition simultanée et de compilation de code .tex. Par la suite, j'ai installé TeXlive distrib et TexMaker sur Arch-Linux afin de pouvoir éditer le code sans connexion internet.

Durant ce TPE, nous avons rencontré très peu de difficultés. Une des principales a été de trouver des papiers en rapport avec les algorithmes génétiques et qui fournissent des explications relativement intuitives de ces fondements afin qu'on soit capable de les expliquer, l'autre difficulté a été de faire en sorte que mon partenaire maîtrise les idées présentes dans ce TPE.

Ce TPE a été l'occasion de grandement me familiariser avec LaTeX(utilisation de macros, de nombreuses balises et extensions (sty) telles que BibTeX, Algorithm2ee, etc) puisque je suis celui qui a écrit tout le code LaTeX. Il m'a aussi permis d'améliorer l'algorithme génétique que j'avais écrit en 3ème (modification de la fonction de fitness, optimisation du code et tentative d'utilisation du multi-threading). Je me suis également intéressé aux techniques d'interpolations polynomiales telles que les interpolations de Lagrange ou encore les polynômes de Tchebychev, je suis allé jusqu'à formaliser la généralisation des interpolations de Lagrange dans $\mathbb{R}^{n\in\mathbb{N}}$ par défi personnel.

L'utilisation du journal de bord m'a paru complètement obsolète à l'époque où nous pouvons stocker, à n'importe quel moment, grâce à nos smartphones, des quantités importantes de données. J'ai énormément écrit chez moi et la tenue de ce carnet ne me paraissait pas être un bon emploi de mon temps, j'ai tout de même essayé de tenir à jour celui-ci. Tous les documents que j'ai pu trouver intéressants et utiles à ce TPE sont disponibles dans les références, de plus, le support informatique est bien plus approprié pour les références URL.

Ce travail a été pour moi un grand plaisir puisque réunissant mes grandes passions, j'ai notamment adoré comprendre comment les chercheurs ont pu arriver à certaines conclusions et expliquer de manière intuitive ce que j'avais compris. Il m'a permis de me poser dans la lecture de plusieurs ouvrages scientifiques d'un certain niveau (disponibles sur ArXiV ou sur d'autres sites de publications participatifs) en démystifiant certains principes et affirmations mathématiques, incompréhensibles au premier regard.

Synthèse de TPE (Killian Henger) : Les algorithmes génétiques.

Le sujet que proposait Yvann regroupait mathématiques et potentiellement physique dans les applications, matières que j'apprécie, de plus, on s'entend très bien dans notre passion pour les sciences.

Nous avons proposé une problématique, au départ, générale, pour ensuite l'affiner à la demande des professeurs. L'apprentissage-machine étant un sujet bien trop large, il nous aurait été aussi impossible de traiter correctement ce sujet en aussi peu de temps. Nous nous sommes montrés très actifs au début du TPE, l'introduction faisant intervenir la génétique m'enthousiasmait mais plus nous rentrions dans le sujet et plus il se complexifiait. Mes connaissances limitées en mathématiques m'ont alors posé problème.

J'ai proposé de montrer des applications des algorithmes génétiques, notamment de celui codé par Yvann, sur des problèmes en rapport avec la physique. Je me suis aussi essayé à manier LaTeX, un tout nouveau outil pour moi, n'étant pas habitué à la syntaxe très algorithmique de cet outil, j'ai décidé de laisser Yvann écrire le code LaTeX, qui lui parraissait, plutôt à l'aise avec. On s'est rendu compte qu'aborder directement la physique était trop éloigné du sujet, or j'avais déjà réalisé cette partie et ai tout supprimé, ainsi, petit à petit, je me suis désintéressé du sujet bien que je continuais de me tenir au courant de l'avancement du TPE. Si je n'ai pas beaucoup apporté à l'écrit, j'ai préparé l'oral comme il le fallait, c'est-à-dire que j'ai fait entièrement la présentation, le plan de l'oral, tout en vérifiant que l'exposé était assez vulgarisé pour que vous, professeur, n'étant pas forcément expert dans ce domaine, puissiez comprendre, ainsi évitant que vous vous sentiez largué comme j'ai pu l'être mais aussi pour qu'il en soit agréable, la vulgarisation n'étant pas forcément un domaine dans lequel Yvann est à l'aise. J'ai également veillé à ce que la rédaction se fasse sans faute de français.

Au final ce TPE m'a été enrichissant en terme de connaissances mais je n'ai malheureusement pas été assez impliqué dans la production finale de celui-ci.

References

[1] P. Pierre Lindenbaum. *Darwin's evolution : four days later.* blogsplot, 2008. URL: http://plindenbaum.blogspot.fr/2008/12/darwins-evolution-four-days-later.html.

[2] P. Bagley J. D. *The Behavior of Adaptive Systems Which Employ Genetic and Correlative Algorithms.* University of Michigan, Ann Arbor, 1967. URL: https://deepblue.lib.umich.edu/handle/2027.42/3354.

[3] H. J. H. H. K. J. N. R. E. and T. P. R. *Induction: Processes of Inference, Learning, and Discovery.* Computational Models of Cognition and Perception. The MIT Press, Cambridge, MA, 1986. URL: http://dl.acm.org/citation.cfm?id=6677.

[4] J. H. Holland. *Adaptation in Natural and Artificial Systems.* first MIT Press ed. The MIT Press Cambridge, MA, 1992. First edition: University of Michigan Press, 1975. URL: http://dl.acm.org/citation.cfm?id=531075.

[5] Contributeurs. *Mathematical optimization.* Wikipedia, 2016.

[6] Contributeurs. *Optimisation combinatoire.* Wikipedia, 2016.

[7] J. S. A.E Eiben. *Introduction to Evolutationary Computing by A.E Eiben and J.E. Smith.* Vrije Universiteit Amsterdam, 2003. URL: http://link.springer.com/book/10.1007%2F978-3-662-44874-8.

[8] A Geyer Schulz. *Fuzzy Rule-Based Expert Systems and Genetic Machine Learning; vol.3 of Studies in Fuzziness.* Physica-Verlag, Heidelberg, 1995.

[9] D. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, Reading, MA, 1989. URL: https://goo.gl/1dCDWj.

[10] J. H. Holland. *Hidden Order: How Adaptation Builds Complexity.* Helix Books, 1996. URL: http://dl.acm.org/citation.cfm?id=225764.

[11] GAUL. *Why does a GA work?* GAUL, 2000,2009. URL: http://gaul.sourceforge.net/why_ga_work.pdf.

[12] C. C. Darrell Whitley Rajarshi Das. *Tacing primary hyperplane competitors during genetic search.* J.C. Baltzer A.G. Scientific Publishing Company, 1992. URL: www.cs.colostate.edu/pubserv/pubs/Whitley-genitor-1991-foga90.ps.gz.

[13] A. H. Wright. *The exact schema theorem.* Arxiv, 2000. URL: https://arxiv.org/abs/1105.3538.

[14] Gurobi. *The Traveling Salesman Problem with integer programming and Gurobi.* Gurobi. URL: http://examples.gurobi.com/traveling-salesman-problem.

[15] J. Kubalík. *Genetic Algorithm for the Traveling Salesman Problem: Initialization, crossover, mutation.* cvut. URL: https://cw.fel.cvut.cz/wiki/_media/courses/a4m33bia/cheatsheet_tsp.pdf.

[16] Z. H. Ahmed. *Genetic Algorithm for the Traveling Salesman Problem using Sequential Constructive Crossover Operator.* Al-Imam Muhammad Ibn Saud Islamic University. URL: http://www.ppgia.pucpr.br/~alceu/mestrado/aula3/IJBB-41.pdf.

[17] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* The MIT Press, Cambridge, 1992. URL: https://books.google.fr/books?id=Bhtxo60BV0EC.

[18] K. K. Ross Foley. *A Comparison of Genetic Algorithms using Super Mario Bros.* WORCESTER POLYTECHNIC INSTITUTE, 2015. URL: https://pdfs.semanticscholar.org/4086/534ab684e26761fae415fff694f4c6692ad1.pdf.

[19] R. M. Kenneth O. Stanley. *Evolving Neural Networks through Augmenting Topologies.* The MIT Press, Cambridge, 2002. URL: http://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf,https://www.youtube.com/watch?v=qv6UVOQ0F44.

[20] R. Johansson. *Genetic Programming : Evolution of Mona Lisa*. 2008. URL: https://rogerjohansson.blog/2008/12/07/genetic-programming-evolution-of-mona-lisa/.