ALGORITHMES GENETIQUES

Les algorithmes génétiques, tout comme les réseaux de neurones, s'inspirent d'une métaphore biologique, en l'occurrence les mécanismes de l'évolution darwinienne et de la génétique pour constituer une méthode de recherche et d'optimisation aux propriétés intéressantes.

1. Un bref historique :

Les algorithmes génétiques, dont la popularité n'a cessé de croître au cours de ces dernières années, ont déjà une histoire relativement ancienne. Ils ont été développés par Holland dès 1962 au cours de ses travaux sur les systèmes adaptatifs mais il a fallu attendre 1985 pour qu'ils soient véritablement popularisés par Goldberg qui est désormais considéré comme le père fondateur. Nous nous bornerons à ne présenter que le fonctionnement général des algorithmes génétiques ; des compléments plus détaillés peuvent être trouvés dans le livre de Goldberg : « Genetic Algorithms in search, Optimization & Machine Learning » éd. Addison Wesley.

2. Motivation :

Les algorithmes génétiques sont des méthodes de recherche combinatoire et d'optimisation qui visent à trouver un compromis entre les deux objectifs souvent contradictoires que l'on rencontre fréquemment en optimisation. A savoir, explorer les meilleures solutions de manière globale tout en convergeant efficacement vers le minimum (ou le maximum) de manière locale. Les méthodes classiques d'optimisation (Newton-Raphson, Simplexe, Gradient) fournissent généralement des très bonnes solutions de manière locale en recherchant la ligne de plus grande pente dans le voisinage du point courant et présentent le risque de converger vers un minimum local. A l'opposé, les méthodes de recherche aléatoire ou les méthodes énumératives sont utiles sur des problèmes très simples mais souvent impossibles à mettre en œuvre et peu efficaces dès lors que l'espace de recherche est grand.

3. L'algorithme génétique simple :

Un peu plus concrètement, un algorithme génétique (AG) est une méthode itérative de recherche globale dont le but est d'optimiser une fonction définie par l'utilisateur. En optimisation, on l'appelle fonction de coût, de profit ou critère, dans le jargon des AG, on l'appellera : fonction d'adaptation. Pour atteindre cet objectif, l'algorithme examine en parallèle l'adaptation d'une population de points, que l'on appellera individus. Chaque individu représente un point de l'espace de recherche et est codé par une chaîne de caractéristiques ou gènes qui peuvent prendre plusieurs valeurs ou allèles. Dans sa forme la plus simple, les seuls allèles possibles sont 0 ou 1 : un individu n'est rien d'autre qu'une chaîne de bits. Le but étant de trouver la combinaison des valeurs possible qui fournisse l'adaptation maximale. A chaque itération ou génération, une nouvelle population de même taille est créée et consistera en des

individus généralement mieux adaptés que la génération précédente. La création de la nouvelle génération s'exécutera généralement en trois étapes :
1. Sélection / Reproduction, 2. Croisement, 3. Mutation.

Lors de la reproduction, les individus sont recopiés en fonction de leur valeur d'adaptation. Les individus les mieux adaptés ont une plus grande probabilité de contribuer à la génération suivante. De manière algorithmique, l'opérateur de reproduction est généralement implanté par une roulette biaisée. Chaque individu se voit attribuer sur la roulette un secteur proportionnel à la contribution relative de son adaptation. On effectue autant de tirages qu'il y a d'individus et les individus tirés au sort sont répliqués de manière exacte pour constituer l'ensemble d'individus à croiser. Chaque individu i, dont l'adaptation est $f(i)$ a une probabilité $f(i)/\sum_j f(j)$ d'être sélectionné.

Pour une population de taille n, l'espérance mathématique du nombre d'enfants d'un individu i est : $n.f(i)/\sum_j f(j)$

Une façon de procéder consiste à sélectionner deux individus par une roulette biaisée et à échanger une partie de leurs gènes. Plus schématiquement, supposons que l'on ait tiré les individus suivants de notre population : P1 = 0111010|11010 et P2 =1010011|10100 où le symbole de séparation | représente un site de croisement, également tiré au sort parmi GENOME_LENGTH -1 où GENOME_LENGTH est la longueur de la chaîne représentant le chromosome d'un individu.
Après croisement, on forme un individu Enfant E = 0111010|10100 formé des premiers gènes de P1 et des derniers gènes de P2.
Dans une version plus élaborée, le croisement des parents n'est effectuée qu'avec une probabilité p_c. On tire un nombre aléatoire p dans [0, 1[, si p< p_c, le croisement a lieu sinon il est rejeté.
Une fois créés, les gènes de l'enfant E peuvent subir des mutations avec une probabilité p_m. Cela consiste simplement à remplacer un gène de l'individu par une autre valeur de l'alphabet des allèles. Dans le cas binaire, le gène choisi aléatoirement pour la mutation sera remplacé par son contraire 0 ou 1.
On remplace finalement l'individu le moins bien adapté de la population par l'enfant E' muté, pour former une nouvelle population de la génération suivante.

D'un point de vue algorithmique, les mécanismes de reproduction, de croisement et de mutation sont extrêmement simples et ne nécessitent que de générer des nombres aléatoires, de copier des chaînes ou d'échanger des morceaux de chaînes entre elles ou d'inverser des bits. L'algorithme génétique simple est paramétré par trois valeurs :

- La taille n de la population ou nombre d'individus
- Le taux de croisement p_c
- Le taux de mutation p_m.

Il est souvent délicat de trouver les bonnes valeurs de ces 3 paramètres car elles dépendent beaucoup du problème considéré. Le nombre d'individus dépend fortement de la longueur GENOME_LENGTH du chromosome. Pour les taux de croisement et de mutation, des valeurs de 0.7 et 0.05 sont fréquemment utilisées.

Au fil des générations, l'adaptation moyenne de la population augmente et la diversité des individus diminue. On supervise en général l'adaptation moyenne et l'adaptation du meilleur individu lors du déroulement de l'algorithme. Plusieurs critères d'arrêt peuvent être choisis suivant la classe du problème. On peut simplement décider d'arrêter l'algorithme au bout d'un nombre de générations donné, ou lorsque la diversité de la population passe sous un certain seuil – un pourcentage donné de la population est regroupé dans une hypersphère de rayon donné – ou bien encore lorsque l'amélioration de l'adaptation du meilleur individu de la population passe sous un certain seuil tout en étant positive.

4. Optimisation d'une fonction simple :

Recherche du maximum de f(x)=4x(1-x) sur [0,1[

Dans cet exemple, volontairement simple, la première opération consiste à choisir un codage des données. On choisira de multiplier x par 256 et de prendre la partie entière du résultat. Chaque x de l'intervalle considéré se verra représenté par une séquence de 8 bits pour le coder dans l'espace de recherche ainsi discrétisé. Par exemple, x=0.26 donnera 66 qui codé en binaire donne : 01000010.
La fonction d'adaptation sera simplement la fonction à maximiser f(x) car f(x) > 0.
Il ne reste plus qu'à appliquer l'algorithme génétique simple. Générons d'abord de manière aléatoire notre population initiale de 4 individus. C'est en général une petite population, sur un cas réel on a une population d'au moins 50 individus. Dans le tableau 1 est regroupée la population initiale, l'adaptation de chacun des individus qui la constitue et leur chance respective de reproduction.

Individu	Valeur x	Adaptation f(x)	Probabilité de reproduction	Probabilités cumulées
10010010	0.5703	0.9802	0.288	0.288
01011101	0.3632	0.9251	0.272	0.560
00100110	0.1484	0.5055	0.148	0.708
10001001	0.5351	0.9951	0.292	1
		Somme=3.4059		

1. Etape de sélection / reproduction :

On tire un premier nombre aléatoire, disons 0.35. En comparant ces valeurs avec la colonne de probabilités cumulées du tableau ci-dessus, on sélectionne l'individu dont la probabilité cumulée est immédiatement supérieure à la valeur tirée au hasard. L'individu 2 est le premier sélectionné. On le nomme Parent1.
On tire un second nombre aléatoire, disons 0.75. L'individu 4 est le second sélectionné. On le nomme Parent2.

2. Etape de croisement :

Lors de l'étape suivante dite de croisement, les deux parents sélectionnés vont être croisés. On choisit alors un site (gène) de croisement aléatoirement, disons 4 parmi les sites disponibles (8 dans notre exemple) pour créer un individu Enfant E.
Avec P1=0101|1101 et P2=1000|1001, on crée E=0101|1001 qui est formé des 4 premiers gènes de P1 et des 8-4 derniers gènes de P2.

3. Etape de mutation :

Cette étape est essentielle pour mieux explorer le domaine de définition de f(x).
On parcourt tous les gènes de l'enfant E et on permute chacun de ses gènes avec une probabilité de mutation p_m.
Supposons désormais que le sort ait décidé de muter les gènes 2 et 7. L'enfant E=0[1]0110[0]1 sera transformé en E'=0[0]0110[1]1
On remplace finalement l'individu 3 le moins bien adapté de la population initiale par l'enfant E' muté, pour former une nouvelle population de la génération suivante.

Individu	Valeur x	Adaptation f(x)	Probabilité de reproduction	Probabilités cumulées
10010010	0.5703	0.9802	0.2990	0.2990
01011101	0.3632	0.9251	0.2822	0.5812
00011011	0.1055	0.3774	0.1151	0.6963
10001001	0.5351	0.9951	0.3036	1
		Somme=3.2778		

En recommençant de manière itérative les phases de sélection/ reproduction, de croisement et de mutation, on tendra vers une population en moyenne mieux adaptée. L'un des individus (voire plusieurs) est 10000000 correspondant à x=0.5 qui est le maximum de la fonction f(x) ; il pourra cependant y avoir des mutants pour préserver la diversité de la population.
Nous venons de décrire le fonctionnement de l'algorithme génétique de base, mais il existe de nombreuses autres variantes. Bien souvent, c'est l'utilisateur lui-même qui concevra son algorithme en fonction de son application propre. En modifiant les opérateurs de sélection, de reproduction ou de mutation pour y introduire sa connaissance du problème pour accélérer sa recherche ou améliorer son efficacité.

5. Algorithme de principe :

- **Initialisation de la population :**
  - Générer aléatoirement une population de `POPULATION_SIZE` individus.
  - Pour chaque individu, initialiser ses gènes avec des valeurs binaires aléatoires.
- **Boucle principale (Générations) :**
  - Répéter les étapes suivantes pour un nombre maximum de générations (`MAX_GENERATIONS`).
- **Sélection des parents et création d'une nouvelle population :**
  - Pour chaque individu de la population existante :
    - Sélectionner deux parents par roulette biaisée (fonction `selectParent`).
    - Créer un nouvel individu (enfant) en utilisant le croisement (crossover) des deux parents (fonction `crossover`).
    - Appliquer une mutation à l'enfant (fonction `mutate`).
    - Calculer la fitness de l'enfant (fonction `calculateFitness`).
    - Remplacer l'individu le moins adapté de la population précédente par le nouvel enfant.
- **Remplacement de l'ancienne population par la nouvelle :**
  - Mettre à jour la population existante avec la nouvelle population créée.
- **Affichage de la meilleure fitness de la génération :**

- Calculer et afficher la meilleure valeur de fitness de la génération en cours.

- **Affichage du meilleur individu de la population finale :**
  - Trouver l'individu avec la valeur de fitness la plus élevée dans la population finale.
  - Afficher la valeur de fitness et les gènes du meilleur individu.

6. Travail à réaliser :

On vous propose dans un premier temps d'utiliser la structure de données suivante :

```
#include <stdint.h>
#define GENOME_LENGTH 64
typedef struct {
    uint32_t genes[GENOME_LENGTH];
    double fitness;
} Individual;
```

pour décrire un individu de votre population. L'avantage d'utiliser un tableau statique dans une structure permet d'utiliser l'opérateur = pour faire une affectation avec copie profonde des données.

Puis dans une version plus élaborée, pour stocker les bits de manière compacte, vous utiliserez la structure :

```
typedef struct {
    uint32_t genes[GENOME_LENGTH / 32];
    double fitness;
} Individual;
```

On vous donne les fonctions suivantes pour accéder à la valeur d'un gène

```
int getGene(Individual *individual, int index) {
    int wordIndex = index / 32;
    int bitIndex  = index % 32;
    return (individual->genes[wordIndex] >> bitIndex) & 1;
}
```

et pour lui affecter une valeur :

```
void setGene(Individual *individual, int index, int value) {
    int wordIndex = index / 32;
    int bitIndex  = index % 32;
    individual->genes[wordIndex] = (individual->genes[wordIndex] &
~(1U << bitIndex)) | (value << bitIndex);
}
```

Votre travail consiste à programmer cet algorithme pour rechercher le minimum d'une fonction d'une variable : x^2 , x^2-40.cos(x), x^2+40.sin(x)

Vous pourrez ensuite traiter aisément le cas d'une fonction de plusieurs variables, en reproduisant le mécanisme présenté ci-dessus pour chaque variable, la phase de sélection restant inchangée.

7. Applications

On vous propose dans un second temps d'appliquer votre code d'algorithme génétique sur l'un des problèmes concrets suivants.

Problème 1 : le voyageur de commerce.
Le problème du voyageur de commerce (TSP) est un défi classique en optimisation combinatoire. Il consiste à trouver le parcours le plus court qui permet à un voyageur de visiter toutes les villes données **une seule fois** et de revenir à son point de départ. Mathématiquement, cela revient à minimiser la distance totale parcourue. Bien que cela puisse sembler simple, le nombre de combinaisons possibles augmente rapidement avec le nombre de villes, rendant la recherche d'une solution optimale très difficile pour un grand nombre de villes. Le TSP a des applications dans de nombreux domaines tels que la logistique, la planification de circuits électroniques et la génomique. Des algorithmes efficaces ont été développés pour trouver des solutions approximatives au problème, mais trouver une solution optimale reste un défi pour un grand nombre de villes.

Problème 2 : optimisation du recouvrement partiel dans un réseau d'antennes GSM en utilisant les algorithmes génétiques.
On désire optimiser le recouvrement partiel d'un réseau d'antennes GSM en utilisant des algorithmes génétiques. Vous disposez d'un ensemble initial d'antennes réparties dans une région géographique donnée. L'objectif est d'ajouter de nouvelles antennes de manière stratégique afin d'améliorer la qualité du service sans redondance excessive.

Le problème est défini comme suit :

- Vous disposez d'une carte représentant la région géographique à couvrir. Définir des cas tests pour lesquels la solution optimale est connue.
- Chaque antenne a une portée spécifique qui détermine sa zone de couverture.
- L'objectif est de maximiser la couverture totale de la région tout en minimisant le recouvrement entre les antennes existantes et nouvellement ajoutées.

La solution optimale peut être une approximation en raison de la complexité du problème. L'objectif principal est de comprendre le processus d'optimisation avec les algorithmes génétiques et d'expérimenter avec différentes configurations pour trouver des compromis satisfaisants entre la couverture et le recouvrement.

Opérateur « non » bit à bit :	~x
Opérateur « et » bit à bit :	x & y
Opérateur « ou » bit à bit :	x | y
Opérateur décalage à gauche :	x<<n 1111 1010 1111 0110 << 6 = 1011 1101 1000 0000

Application : extraire les 10 premiers bits d'une expression codée sur 16 bits (4 octets) :

```
x       =1010 0111 0110 0110 ;
mask    =1111 1111 1111 1111 ;
mask=mask<<6 ;                  /* mask        =1111 1111 1100 0000  */
y=x & mask ;                    /* y           =1010 0111 0100 0000  */
```

Application : extraire les 6 derniers bits d'une expression codée sur 16 bits (4 octets) :

```
x       =1010 0111 0110 0110 ;
mask    =1111 1111 1111 1111 ;
mask=mask<<6 ;                  /* mask        =1111 1111 1100 0000  */
                                /* ~mask       =0000 0000 0011 1111  */
y=x & (~mask) ;                 /* y           =0000 0000 0010 0110  */
```