CHAPITRE 21

Algorithmes évolutionnaires

Nicolas Durand – Jean-Marc Alliot

21.1 Introduction

Les algorithmes évolutionnaires sont des algorithmes d'optimisation s'appuyant sur des techniques dérivées de la génétique et de l'évolution naturelle[1] : croisements, mutations...

Encore peu connus en France[2], leur popularité est croissante aux États-Unis surtout depuis 1985.

Ce sont des méthodes d'optimisation stochastique qui ne requièrent *a priori* pas de régularité sur les fonctions optimisées : les applications possibles sont très diverses. Ils ont la propriété de pouvoir localiser plusieurs optima mais présentent deux inconvénients majeurs. D'une part, ce sont des méthodes chères en calculs qui ne garantissent pas l'optimalité du résultat et l'on conseille généralement d'appliquer ce type de méthodes lorsque *rien d'autre ne marche*. D'autre part, la mise en oeuvre pratique de ces méthodes demande une grande expérience et les résultats théoriques sont généralement peu utiles en pratique.

On distingue quatre familles historiques :

- les algorithmes génétiques sont apparus aux Etats-Unis dans les années 60 : les premiers travaux de John Holland sur les systèmes adaptatifs remontent à 1962 (Holland 1962) ;
- la programmation génétique a d'abord été considérée comme un sous-domaine des algorithmes génétiques, c'est en train de devenir un domaine de recherche à part entière. Son défi est *d'apprendre à la machine à exécuter des tâches sans qu'elle ait été explicitement programmée pour cela*.
- Les stratégies d'évolution sont nées en Allemagne dans les années 60 (Fogel 1962) pour résoudre des problèmes d'optimisation numérique.
- La programmation évolutionnaire, apparue dans les années 60 en Californie, d'abord appliquée sur des machines à états finis.

---

1 Il est intéressant de trouver dans l'œuvre d'un spécialiste de zoologie, Richard Dawkins (Dawkins 1989), un exemple informatique tendant à prouver la correction de l'hypothèse darwinienne de la sélection naturelle. La méthode utilisée est presque semblable aux techniques d'algorithmes génétiques.

2 Dans (Jones 1992), on trouve une présentation des techniques auto-évolutives, un sujet assez proche des algorithmes génétiques.

21.2 Algorithmes génétiques

21.2.1 Principe général

Un algorithme génétique réalise une optimisation dans un espace de données. Pour pouvoir utiliser les techniques génétiques il faut disposer de deux opérations :

1. une fonction de codage de la ou des données en entrée : originellement les données étaient codées sous forme d'une séquence de bits, d'autres forme de codages sont utilisées en pratique (voir paragraphe 21.5.1) ;
2. fournir une fonction d'utilité, d'adaptation ou *fitness* $U(x)$, permettant de calculer l'adaptation d'une séquence de bits x. Idéalement, cette fonction vaudra 1 si la séquence de bits est parfaitement adaptée et 0 si la séquence est inadaptée. Cette fonction d'utilité correspond au critère à maximiser sur l'espace de données.

Ces deux éléments sont les seuls éléments spécifiques au problème à résoudre. Une fois qu'ils sont fixés, l'algorithme génétique que l'on appliquera sera toujours le même, que l'on optimise la distribution de gaz dans un pipeline, la trajectoire de deux mobiles qui doivent s'éviter, ou la fonction énergie d'un réseau de neurones.

Il s'exécutera en plusieurs étapes :

1. génération aléatoire d'un certain nombre de séquences de bits pour composer la « population » initiale ;
2. mesure de l'adaptation de chacune des séquences présentes ;
3. reproduction de chaque séquence en fonction de son adaptation. Les séquences les mieux adaptées se reproduisent mieux que les séquences inadaptées. La nouvelle population est composée des séquences après reproduction ;
4. on remplace un certain nombre de paires de séquences tirées aléatoirement par le croisement de ces paires (le lieu de croisement dans la séquence de bits est également choisi de façon aléatoire.). Chaque nouvelle paire est constituée de la façon suivante :
   - une séquence est composée de la première partie de la première séquence et de la seconde partie de la seconde séquence ;
   - l'autre séquence est composée de la première partie de la seconde séquence et de la seconde partie de la première séquence.
5. mutation d'un bit choisi aléatoirement dans une ou plusieurs séquences tirées au sort ;
6. Retour à l'étape 2.

Nous allons voir, dans le détail et sur un exemple, comment fonctionne un algorithme génétique.

21.2.2 Un exemple

Nous allons considérer l'exemple suivant : soit la fonction[3] $f(x) = 4x(1-x)$ prenant ses valeurs sur $[0, 1[$. Nous souhaitons trouver le maximum de cette fonction.

---

3 Cet exemple n'est certainement pas le meilleur que l'on puisse prendre, pour bien des raisons. Goldberg présente quant à lui la maximisation de x^2, mais il nous a semblé plus intéressant de trouver le minimum d'une fonction dont l'extremum n'est pas une borne de l'intervalle.

Séquence	Valeur	$U(x)$	% de chance de reproduction	% cumulés	Après reproduction
10111010	0.7265625	0.794678	0.794678 / 2.595947 = 0.31	0.31	11011110
11011110	0.8671875	0.460693	0.460693 / 2.595947 = 0.18	0.31+0.18=0.49	10111010
00011010	0.1015625	0.364990	0.364990 / 2.595947 = 0.14	0.49+0.14=0.63	01101100
01101100	0.4218750	0.975586	0.975586 / 2.595947 = 0.37	0.62+0.37=1.00	01101100
=		2.595947			

Table 21.1 – Population initiale et adaptation au milieu

Pour coder les données, on peut se contenter de multiplier x par 2^8 et de prendre la partie entière du résultat. On obtient ainsi un nombre entier appartenant à l'intervalle $[0, 2^8 - 1]$. Sa représentation en binaire donne une séquence de 8 bits[4] le codant. Ainsi $x = 0.3$ multiplié par 2^8 donne 76.8, dont on prend la partie entière : 76. 76 en binaire donne : 01001100. C'est la séquence de bits codant 0.3.

La fonction $U(x)$ est ici : $U(x) = f(x)$. Il s'agit donc exactement de la fonction à maximiser.

Les deux fonctions étant fixées, il ne reste qu'à appliquer l'algorithme génétique.

Première étape : nous commençons par générer aléatoirement les séquences qui vont composer notre population. Nous ne générons que 4 séquences pour cet exemple particulier[5]. Nous présentons dans le tableau 21.1 la population initiale générée, ainsi que la valeur de la fonction d'adaptation et le pourcentage de reproduction pour chaque séquence dans les trois premières colonnes.

Nous devons ensuite effectuer la deuxième étape, la reproduction. Pour savoir quels sont les quatre éléments de la population suivante avant croisement, on tire aléatoirement quatre nombres entre 0 et 1. La comparaison du nombre avec le pourcentage cumulé de la quatrième colonne du tableau 21.1 permet de savoir quel est l'élément généré. Si nous tirons les 4 nombres 0.47, 0.18, 0.89, 0.75, nous obtenons la colonne 6 du tableau comme population après reproduction.

Passons maintenant au croisement. Un certain nombre de séquences de la population vont être remplacées par des croisements avec d'autres membres de cette population. Typiquement, de 25 à 75% des séquences sont laissées intactes, les autres étant remplacées par des croisements. Supposons que le sort désigne les séquences 1 et 3 comme devant être croisées et remplacées par les résultats du croisement. Il faut encore tirer aléatoirement un nombre entre 1 et 7 qui va désigner l'endroit où le croisement va s'effectuer ; supposons que ce soit 5. Le croisement est alors représenté sur la figure 21.1.

La population après croisement est représentée dans le tableau 21.2. Il resterait à appliquer les mutations. Mais le pourcentage de mutation est toujours très faible (inférieur ou très inférieur à 10%) et nous supposons qu'il n'y en a pas ici.

Nous constatons que non seulement la moyenne des valeurs d'adaptation est meilleure, mais que la qualité de la meilleure séquence est plus grande.

---

4 Nous aurions pu multiplier par 2^n et nous aurions alors obtenu une séquence de n bits. La valeur de n que l'on choisit dépend, bien entendu, de la précision que l'on souhaite obtenir.

5 Ceci est très peu. Dans un cas réel, la population est beaucoup plus importante ; plusieurs centaines de séquences est un nombre raisonnable.

$$\begin{array}{r} 01101|100 \\ 11011|110 \\ \hline 01101|110 \\ 11011|100 \end{array}$$

Figure 21.1 – Croisement des chaînes 3 et 1

Après croisement	Valeur	$U(x)$
01101110	0.4296875	0.980225
10111010	0.7265625	0.794678
11011100	0.8593750	0.483398
01101100	0.4218750	0.975586

Table 21.2 – Population finale

21.2.3 Discussions des divers paramètres

Comme souvent pour ce type de mécanisme qui pourrait apparaître comme un peu « magique » (les réseaux de neurones font partie de cette catégorie), les difficultés sont invisibles, mais bien réelles. La première d'entre elles consiste à bien choisir les divers paramètres : pourcentage de croisement, pourcentage de mutation, taille de la population...

Mais le problème le plus important est le codage des données. Comme le dit Goldberg lui-même, le codage des données est un art, et de cet art dépend le succès ou l'échec de la tentative. Dans un cas aussi simple que celui-ci, le codage parait évident et pourtant, on remarquera que les séquences 10000000 et 01111111 sont très bien adaptées l'une et l'autre, bien que leur codage soit fondamentalement différent, ce qui n'est pas une bonne propriété[6]. La théorie des graphes présentée dans la section 21.3.2 permet de comprendre ce qu'est un bon codage de données.

21.3 Convergence théorique

21.3.1 Généralités

Plusieurs approches ont été envisagées. La théorie des schémas présentée dans la section 21.3.2 est la plus simple à aborder et donne dans un cadre simplifié une idée du mécanisme de convergence des AGs. On trouve néanmoins plusieurs démonstrations de convergence théorique des algorithmes évolutionnaires dont :

– Un résultat très général de convergence globale proposé par Zhigljavsky (Zhigljavsky 1991), au sens de la convergence faible des mesures de probabilités.

6 Une technique classique, déjà rencontrée pour les ALN, consiste à utiliser des vecteurs d'un espace de Hamming (Schraudolf and Grefenstette 1991).

– Des résultats partiels basés sur un modèle des algorithmes évolutionnaires en tant que chaînes de Markov dans l'espace des populations (Davis and Principe 1991; Davis and Principe 1993; Nix and Vose 1992; Fogel *et al.* 1966).
– Un résultat de convergence fort pour les AGs basé sur la théorie de Friedlin Weinzel des perturbations stochastiques des systèmes dynamiques (Cerf 1994). Cette dernière approche est la plus satisfaisante tant sur le plan mathématique, que sur celui de la modélisation, les différents opérateurs étant présentés comme "perturbant" un processus Markovien représentant la population à chaque étape. Ici encore il est démontré l'importance de l'opérateur de mutation, le croisement pouvant être totalement absent. L'investissement nécessaire pour la compréhension des démonstrations de Cerf est assez important, le lecteur intéressé se reportera aux références citées. De plus, de nombreuses interrogations demeurent cependant concernant les relations réelles entre les différents paramètres caractérisant l'algorithme génétique et les choix pratiques de ceux-ci. Dans ce domaine, la pratique devance encore la théorie, même si les mécanismes commencent à être plus clairs. Les raffinements utilisés en pratique pour faire converger l'AG (sharing, scaling, clustering) n'ont pas encore été étudiés sur le plan théorique.
– Des résultats de convergence globaux avec taux de convergence pour certains algorithmes de stratégies d'évolution, et sur des fonctions convexes (Back *et al.* 1993). Le fait que ces résultats soit obtenus sur des fonctions convexes appauvrit largement leur intérêt, sachant que pour de telles fonctions, des algorithmes classiques de type gradient sont généralement largement plus performants.

21.3.2 Théorie des schémas

Définitions fondamentales

Nous allons d'abord poser quelques définitions :

Définition 21.1 – Séquence – *On appelle séquence A de longueur $l(A)$ une suite $A = a_1 a_2 \dots a_l$ avec $\forall i \in [1, l], a_i \in V = \{0, 1\}$.*

Ceci correspond à la notion de séquence de bits que nous avons utilisée jusqu'à présent.

Nous avons également besoin de la notion de schéma et de quelques définitions associées :

Définition 21.2 – Schéma – *On appelle schéma H de longueur l une suite $H = a_1 a_2 \dots a_l$ avec $\forall i \in [1, l], a_i \in V^+ = \{0, 1, *\}$.*

Comme nous allons le voir, une $*$ en position i signifie que a_i peut être indifféremment un 0 ou un 1.

Définition 21.3 – Instance – *On dit qu'une séquence $A = a_1 \dots a_l$ est une instance d'un schéma $H = b_1 \dots b_l$ si pour tout i tel que $b_i \neq *$ on a $a_i = b_i$.*

Ainsi, $H = 010 * 0101$ est un schéma et les séquences 01000101 et 01010101 sont des instances de H (ce sont même les seules).

Définition 21.4 – Position fixe, position libre – *Soit un schéma H. On dit que i est une position fixe de H si $a_i = 1$ ou $a_i = 0$. On dira que i est une position libre de H si $a_i = *$.*

Définition 21.5 – Ordre d'un schéma – *On appelle ordre du schéma H le nombre de positions fixes de H. On note l'ordre de H $o(H)$.*

Par exemple, le schéma $H = 01**10*1$ a pour ordre $o(H) = 5$, le schéma $H' = *****101$ a pour ordre $o(H') = 3$. On remarquera qu'un schéma H de longueur $l(H)$ et d'ordre $o(H)$ admet $2^{(l(H)-o(H))}$ instances différentes.

Définition 21.6 – Longueur fondamentale – *On appelle longueur fondamentale du schéma H la distance séparant la première position fixe de H de la dernière position fixe de H. On note cette longueur fondamentale $\delta(H)$.*

Ainsi, le schéma $H = 1**01***$ a pour longueur fondamentale $\delta(H) = 5-1 = 4$, le schéma $H' = 1******1$ a pour longueur fondamentale $\delta(H') = 8 - 1 = 7$, et pour le schéma $H\ >>= **1*****$ nous avons $\delta(H\ >>) = 3 - 3 = 0$.

Définition 21.7 – Adaptation d'une séquence – *On appelle adaptation d'une séquence A une valeur positive que nous noterons $f(A)$.*

f est ce que nous avions appelé jusqu'à présent la fonction d'utilité[7].

Définition 21.8 – Adaptation d'un schéma – *On appelle adaptation d'un schéma H la valeur*

$$f(H) = \frac{\sum_{i=1}^{2^{(l(H)-o(H))}} f(A_i)}{2^{(l(H)-o(H))}}$$

où les A_i décrivent l'ensemble des instances de H.

L'adaptation d'un schéma est donc la moyenne des adaptations de ses instances.

Effets de la reproduction

Soit un ensemble $S = \{A_1, \ldots, A_i, \ldots, A_n\}$ de n séquences de bits tirées aléatoirement. Durant la reproduction, chaque séquence A_i se reproduira avec une probabilité :

$$p_i = \frac{f(A_i)}{\sum_{i=1}^{n} f(A_i)}$$

Supposons que nous ayons à l'instant t un nombre $m(H,t)$ de séquences représentant le schéma H dans la population S. À l'instant $t+1$ nous devons statistiquement avoir un nombre :

$$m(H,t+1) = m(H,t).n.\frac{f(H)}{\sum_{i=1}^{n} f(A_i)}$$

Posons

$$\overline{f_t} = \frac{\sum_{i=1}^{n} f(A_i)}{n}$$

$\overline{f_t}$ représente la moyenne de l'adaptation des séquences à l'instant t. La formule précédente devient :

$$m(H,t+1) = m(H,t)\frac{f(H)}{\overline{f_t}}$$

7 Le « f » de f ne vient pas de fonction, mais de l'anglais « fitness ».

Posons $c_t(H) = \frac{f(H)}{f_t} - 1$. On obtient alors :

$$m(H, t+1) = (1 + c_t(H))m(H, t)$$

On voit donc qu'un schéma, dont l'adaptation est au-dessus de la moyenne, voit son nombre de représentants augmenter, suivant une progression qui est de type géométrique si nous faisons l'approximation que $c_t(H)$ est constant dans le temps. Nous avons alors :

$$m(H, t) = (1 + c(H))^t.m(H, 0)$$

Le résultat est donc clair : si la reproduction seule était en jeu, les schémas forts éliminer-aient très rapidement les schémas faibles.

Effets des croisements

Nous allons nous intéresser à la probabilité de survie $p_s(H)$ d'un schéma H lors d'une opération de croisement. Considérons cela sur un exemple. Soit le schéma $H = **10*1**$. Supposons qu'une séquence qui est une instance de ce schéma soit croisée avec une autre séquence. Quelle est la probabilité pour que la séquence résultante soit encore une instance de H ? Il est impossible de répondre exactement à la question, tout au plus peut-on donner une borne inférieure de cette valeur. Il est clair que H ne sera pas détruit si le site de croisement qui est tiré au sort est inférieur à 3 (avant le premier 1) ou s'il est supérieur à 5 (après le dernier 1)[8].

On voit donc immédiatement qu'une borne inférieure de la probabilité de détruire un schéma H est $\delta(H)/(l-1)$. Donc la probabilité de survie dans un croisement est $1 - \delta(H)/(l-1)$. Si d'autre part on ne croise qu'une fraction p_c de séquences dans une population donnée, la probabilité de survie est donnée par :

$$p_s \geq 1 - p_c \frac{\delta(H)}{l-1}$$

En rassemblant ce résultat et celui du paragraphe précédent, nous obtenons pour l'évolution d'une population[9] :

$$m(H, t+1) \geq m(H, t)(1 + c_t(H))(1 - p_c \frac{\delta(H)}{l-1})$$

Effets des mutations

Supposons que la probabilité de mutation d'un bit dans une séquence soit p_m. Dans un schéma H, seules les positions fixes peuvent être détruites. Comme la probabilité de survie d'un bit est $1 - p_m$, la probabilité de survie d'un schéma H contenant $o(H)$ positions fixes est $(1 - p_m)^{o(H)}$. La probabilité de mutation étant toujours petite devant 1, on peut faire un développement limité au premier ordre, ce qui nous donne une probabilité de survie de $1 - o(H)p_m$.

En introduisant ce terme, nous avons donc l'équation finale :

$$m(H, t+1) \geq m(H, t)(1 + c_t(H))(1 - p_c \frac{\delta(H)}{l-1} - o(H)p_m)$$

---

8 Dans tous les autres cas, H peut être (ou ne pas être) détruit.

9 On rappelle que les croisements et les mutations se font sur la population reproduite, et non sur la population initiale.

Conclusions

La formule ci-dessus nous apprend deux choses :
- les schémas qui ont une longueur fondamentale petite sont plus favorisés que les autres, lors de la génération d'une nouvelle population ;
- les schémas qui ont un ordre petit sont plus favorisés que les autres, lors de la génération d'une nouvelle population.

Ceci enseigne une chose fondamentale pour le codage de données : les schémas qui codent les données « intéressantes » pour le problème doivent avoir un ordre et une longueur fondamentale faibles[10], alors que les données « sans intérêt » doivent être codées par des schémas qui ont un ordre et une longueur fondamentale élevés.

Les résultats présentés ci-dessus ne sont qu'une introduction à la théorie des schémas, il existe de nombreux approfondissements. On trouvera dans les références suivantes les principaux modèles théoriques sur la théorie des schémas et ses extensions :(Goldberg 1989b; Vose 1991; Bui and Moon 1993; Goldberg 1989a; Bridges and Goldberg 1987; Caruana and Schaffer 1988; Moon 1994; Dasgupta and McGregor 1992; Smith *et al.* 1991)

21.4 Exemple complet d'application

21.4.1 Principe

Nous allons tester l'utilisation des algorithmes génétiques sur un cas très simplifié d'optimisation de trajectoires d'avions dans le plan horizontal. Pour plus détails, on peut se reporter à (Durand 1996). Nous posons comme hypothèses :
- à chaque avion est assigné un point de départ et d'arrivée situé à 20 minutes de vol du point de départ ;
- à tout instant les avions doivent se trouver à une distance l'un de l'autre d'au moins 15km (on suppose que c'est le cas à $t = 0$) ;
- les n avions naviguent à vitesse constante ;
- chaque avion peut effectuer au plus une altération de cap de 10, 20 ou 30 degrés à droite ou à gauche à un instant t_0. Il prend ensuite un cap direct vers sa destination à un instant t_1 ;
- on cherche à minimiser le nombre de manœuvres données aux avions, et à nombre de manœuvre constant, minimiser le retard moyen des avions à l'arrivée engendré par les manœuvres.

21.4.2 Modélisation du problème

Un avion est ici défini comme un point matériel muni d'un vecteur vitesse (dans l'exemple qui suit, les avions volent à 500 nœuds ou 900 km/heure). Nous nous sommes limités au plan horizontal car, en contrôle aérien, on préfère donner aux avions des manœuvres dans le plan horizontale plutôt que des manœuvres de montée ou de descente

10 Les données intéressantes sont bien entendu les données qui sont proches de la solution. Un bon codage des données implique donc d'avoir une *idée* de la forme de la solution.

(leur coût en termes de consommation et d'exécution est plus faible, le confort du passager est préservé). Le temps est discrétisé par pas de 10 secondes.

21.4.3 Application de l'algorithme génétique

Le codage binaire présenté dans la section 21.2.2 pourrait être utilisé dans le cas présent, mais on lui préférera un codage plus immédiat qui préserve la structure des données. La manœuvre effectuée par un avion est déterminée par un triplet (a, t_0, t_1) où $a \in \{-30, -20, -10, 0, 10, 20, 30\}$, t_0 et t_1 sont des variables réelles comprises entre 0 et 20 minutes. C'est ce triplet qui sera retenu comme codage, les opérateurs de croisement et de mutation devront être adaptés en conséquence. Un problème à n avions sera codé par un $3n$ -uplet.

Le critère d'évaluation retenu s'exprime de la façon empirique suivante :

$$\begin{aligned} f &= \frac{1}{2+C} \quad si\ C > 0 \\ &= \frac{1}{2} + \frac{1}{1+N+R} \quad si\ C = 0 \end{aligned}$$

où C est la durée totale de conflits (somme des durées des périodes pendant lesquelles deux avions sont à moins de 15 km l'un de l'autre), N est le nombre de manœuvres (on considère qu'il y a manœuvre si $a \neq 0$ et $t_0 < t_1$) et R est le retard moyen à l'arrivée relatif à la durée du vol (20 minutes). D'autres choix sont possibles, celui-ci garantie qu'une solution sans conflit ($> \frac{1}{2}$) est toujours plus adaptée qu'une solution avec conflit ($< \frac{1}{2}$). Par ailleurs, N est entier et $R < 1$ donc le critère d'évaluation minimise le nombre de manœuvres avant de minimiser le retard.

Compte-tenu du codage des données choisi, nous devons définir les opérateurs de croisement et de mutation.

L'opérateur de croisement choisi (voir figure 21.2) est le suivant :

- la variable t_0 de l'avion 1 de l'enfant 1 est le barycentre de la variable t_0 de l'avion 1 du parent X affectée du coefficient α et de la variable t_0 de l'avion 1 du parent Y affectée du coefficient $(1-\alpha)$; pour l'enfant 2, on utilise les coefficients $(1-\alpha)$ et α ; ce processus est répété pour la variable t_0 de l'avion 2 et pour les variables t_1 des avions 1 et 2 des deux enfants ; à chaque fois, une valeur différente de α est choisie aléatoirement dans l'intervalle $[-0.5, 1.5]$[11] avec une probabilité uniforme ;
- la variable a de l'avion 1 de l'enfant 1 est choisie aléatoirement avec une probabilité $\frac{1}{2}$ parmi les variables a des avions 1 des deux parents.

L'opérateur de mutation choisi aléatoirement avec des probabilités identiques la variable t_0, t_1 ou a de l'avion 1 ou 2 d'un élément de population et le modifie en lui ajoutant un bruit gaussien (s'il s'agit de t_0 ou t_1) ou en lui ajoutant ou retranchant 10 degrés selon les possibilités (s'il s'agit de a).

La figure 21.3 montre une solution obtenue après 28 générations de l'algorithme génétique. Les paramètres choisis étaient :

---

11 Ne pas prendre α dans $[0, 1]$ mais dans $[-0.5, 1.5]$ permet d'éviter de se restreindre au segment dont les extrémités sont les variables des deux parents.

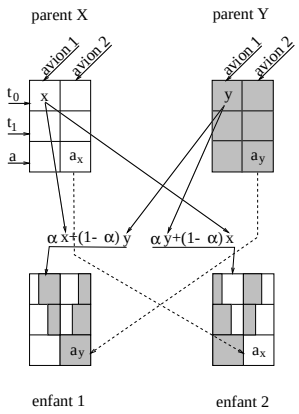

Figure 21.2 – Opérateur de croisement

– population comprenant 100 individus ;
– 60% de croisements à chaque génération ;
– 0.15% de mutations à chaque génération.

La durée d'une génération est alors de l'ordre de 3 centièmes de seconde sur un Pentium II 450. L'optimisation complète dure moins d'une seconde.

Cette solution est en fait optimale (ce résultat ne pouvant être assuré *a priori*, mais seulement constaté a posteriori). Un algorithme de type $A*$, appliqué au problème, fournit un résultat équivalent en un temps équivalent. L'algorithme génétique est donc, *pour cet exemple*, équivalent à une méthode de parcours de graphe.

21.4.4 Conflit à 5 avions

Nous avons ensuite voulu tester l'algorithme dans le cas où le conflit implique 5 avions initialement situés régulièrement sur un demi-cercle de 150 km de rayon et convergent vers le centre du cercle. La taille de la population a été augmentée à 500 individus. Les autres paramètres sont restés inchangés.

La figure 21.4 montre une solution obtenue après 200 générations de l'algorithme génétique. Le temps de calcul est de 1mn 20s.

La solution obtenue n'est pas optimale. Nous verrons par la suite comment on peut l'améliorer en utilisant des méthodes de *sharing* (section 21.5.5) et la *séparabilité partielle* du problème (section 21.5.6).

L'algorithme $A*$ ne permet pas de trouver la solution optimale car son temps d'exécution est trop long. 20mn de vol représentent 120 pas de temps de 10s. Un avion qui n'a pas encore engagé de manœuvre a 7 possibilités de modification de cap à chaque pas de temps (de -30 à 30 degrés par pas de 10). S'il a engagé une manœuvre, il peut soit la poursuivre, soit prendre un cap vers sa destination. Une fois sur la branche de retour, il n'a

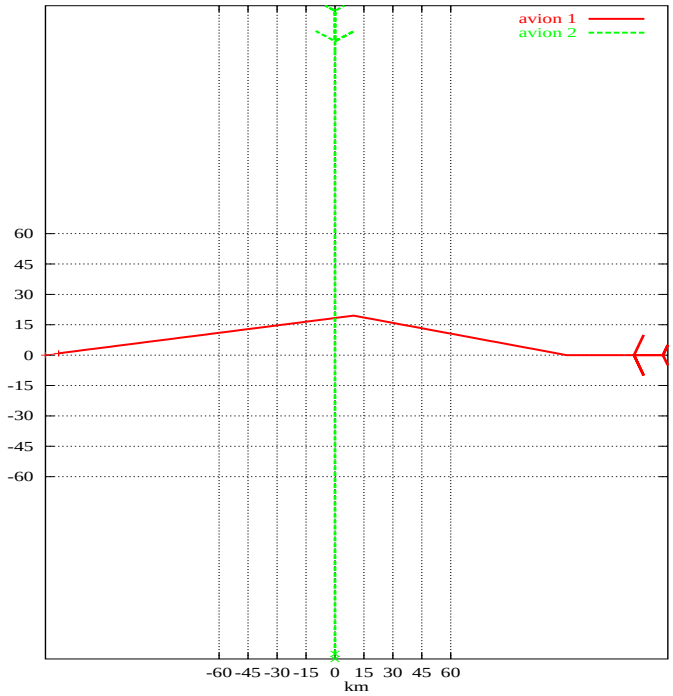

Figure 21.3 – Résultat de l'algorithme génétique (n=28)

plus d'alternative. La taille de l'arbre de recherche pour chaque avion est $7 + 3\,n\,(n+3)$ avec $n = 120$. Si le conflit implique p avions, la taille de l'arbre devient $[7+3\,n\,(n+3)]^p$. Pour 2 avions, la taille reste inférieure à 10^{10}. Pour 5 avions, on dépasse 10^{23}.

21.5 Techniques avancées

Nous allons examiner rapidement dans cette section différentes techniques complémentaires qui peuvent améliorer les performances des algorithmes génétiques. Ces techniques sont pour la plupart présentées en détail dans (Goldberg 1989c), à l'exception des représentations spécialisées, que l'on peut trouver dans (Michalewicz 1992), avec bien d'autres.

21.5.1 Utilisation d'un codage et d'opérateurs spécialisés

Les techniques de codage sous forme de chaînes de bits et de croisement standard présentent un certain nombre d'inconvénients :

- il n'est pas facile de choisir un « bon » codage adapté à la structure du problème ;
- l'application de la fonction de décodage lors de l'évaluation de la fitness est très coûteuse en temps de calcul ;

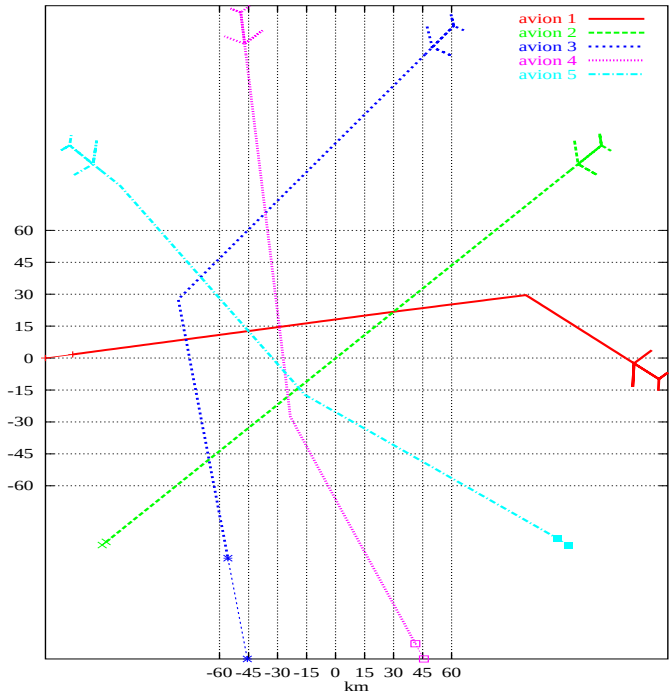

Figure 21.4 – Conflit à 5 avions

– les opérateurs de croisement/mutation sont « aveugles », et ne tiennent aucun compte de la structure du problème.

Il est alors naturel de tenter de définir des codages et des opérateurs spécialisés, qui s'appliqueront efficacement sur le problème considéré.

Dans l'exemple de la section 21.2.2, le codage en chaîne de bits a été abandonné au profit d'un codage utilisant directement les variables du problème. Les opérateurs de croisement et de mutations ont du alors être définis en fonction du codage. Les variables réelles et entières peuvent être traitées différemment. Ce type de représentation a pour but de mieux respecter la topologie de l'ensemble sur lequel l'optimisation est opérée à travers le codage.

Dans le cas des variables réelles, on emploie de façon classique le *croisement barycentrique*. Pour générer deux enfants, on commence par tirer un nombre α au hasard dans l'intervalle $[-0.5, 1.5]$ et par appliquer les formules suivantes :

$$\begin{aligned} p'_1 &= \alpha p_1 + (1-\alpha)p_2 \\ p'_2 &= \alpha p_2 + (1-\alpha)p_1 \end{aligned}$$

En ce qui concerne la mutation, on génère l'élément muté en appliquant la formule :

$$p'_1 = p_1 + B(0, \sigma)$$

où $B(0,\sigma)$ est un bruit gaussien centré en 0 et d'écart type σ. Le choix de σ dépend, bien entendu, du problème considéré.

Il est extrèmement important de réaliser que le codage et les opérateurs choisis conditionnent fortement la qualité de la solution obtenue. Les algorithmes génétiques ne sont pas des boîtes noires miraculeuses dans lesquelles il suffirait d'introduire une fonction pour en retirer l'optimum. Sans une réflexion appropriée sur le codage et les opérateurs, les résultats seront au mieux médiocres.

21.5.2 Opérateurs de réorganisation

L'un des problèmes des algorithmes génétiques est que le sens d'un allèle est lié à sa position, position qui dépend du codage. Ainsi, supposons que dans un premier codage une propriété importante soit codée (sur 2 bits) par le premier et le huitième bit d'une chaîne de 8 bits. Avec cette forme de codage, tout croisement opérera sur cette propriété. Si au contraire la propriété était codée sur le premier et le second bit (par exemple), le bloc serait presque insensible au croisement. C'est de cette constatation que vient l'idée de réorganiser les bits dans un individu, de façon à créer des blocs stables.

Un opérateur de réorganisation opère de la façon suivante : soit l'élément de population 10100010 ; on tire au hasard deux positions, par exemple 3 et 7. Tous les bits compris entre les positions 3 et 7, c'est à dire la séquence 1000, sont alors inversés, pour donner l'individu 10000110.

Se pose alors un premier problème : dans la présentation standard des algorithmes génétiques, le codage d'une propriété dépend de la position de chaque bit. En un mot, l'individu 10100010 et l'individu 10000110 ne code plus le même objet. Il faut donc rendre le décodage indépendant de la position en stockant avec la valeur de chaque bit la valeur de sa position. Ainsi, le premier individu deviendrait $(1,1),(2,0),(3,1),(4,0),(5,0),(6,0),(7,1),(8,0)$ et le second serait alors $(1,1),(2,0),(7,1),(6,0),(5,0),(4,0),(3,1),(8,0)$. Dans ces conditions, les deux individus représentent bien le même objet et seront décodés de la même façon.

Il reste un dernier problème à régler : comment effectuer un croisement ? En effet, considérons les deux individus :

$$(1,1),(2,0),(7,1),(6,0),(5,0),(4,0),(3,1),(8,0)$$

et

$$(1,1),(2,0),(3,1),(4,0),(5,0),(6,1),(7,1),(8,0)$$

Supposons que le site de croisement tiré soit le milieu. Nous aurions alors les deux individus :

$$(1,1),(2,0),(7,1),(6,0),(5,0),(6,1),(7,1),(8,0)$$

et

$$(1,1),(2,0),(3,1),(4,0),(5,0),(4,0),(3,1),(8,0)$$

Aucun de ces individus n'est viable, puisque le premier a deux fois les positions 6 et 7 et qu'il lui manque les positions 3 et 4 et réciproquement pour le second. Il faut donc définir différemment l'opérateur de croisement. Il existe diverses stratégies (voir (Goldberg 1989c)). Nous décrirons brièvement la technique dite de *partially matched crossover*

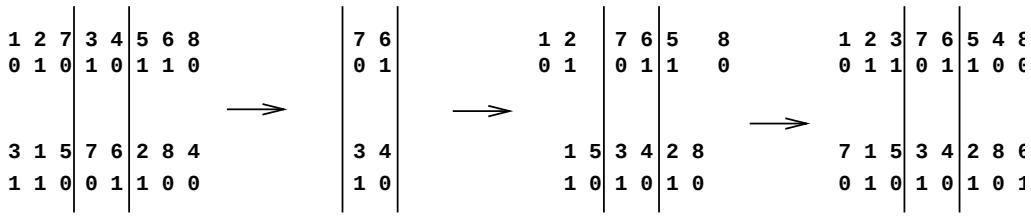

Figure 21.5 – Technique PMX

(PMX). Dans cette technique, on tire deux sites de croisement au lieu d'un seul (voir figure 21.5).

La zone située entre ces deux sites est d'abord échangée : on dit alors que l'allèle 3 *matche* (ou correspond à) l'allèle 7 et que l'allèle 4 matche l'allèle 6 (et réciproquement). Puis, on remet à leur place les allèles qui ne sont pas dupliqués après l'échange des zones de croisement. Dans le cas du premier individu, il s'agit des allèles autres que 7 et 6, et dans le cas du second, des allèles autres que 3 et 4. Puis on remplace dans l'individu du haut ce qui était l'allèle numéro 7 par son correspondant en numéro, c'est à dire l'allèle 3 de l'individu du bas ; on fait de même pour l'allèle 6, et on procède de la même façon pour les allèle 3 et 4 de l'individu du bas.

21.5.3 Le scaling ou mise à l'échelle

Le processus de sélection des individus qui consiste à reproduire en plus grand nombre les individus bien adaptés que les individus mal adaptés conditionne fortement la convergence de l'algorithme. La technique de mise à l'échelle ou *scaling* a pour but de modifier la pression de sélection. Elle consiste à composer la fonction d'évaluation avec une fonction croissante. Considérons l'exemple suivant où les 4 individus X_1, X_2, X_3 et X_4 de la population ont pour adaptation 10, 20, 30 et 40. Si l'on applique les fonctions linéaires croissantes $f \rightarrow \frac{5}{3}(f - 10)$ ou $f \rightarrow 0.2\, f + 20$ sur la valeur de l'adaptation, on donne plus ou moins d'importance aux individus (voir figure 21.6). Dans le premier cas, on augmente la pression de sélection, dans le deuxième cas, on la diminue.

21.5.4 Modes de sélection, tournoi

Le mode de sélection des individus présenté dans la section 21.2.2 se contente de reproduire les individus proportionnellement à leur critère d'adaptation. D'autres modes de sélection sont envisageables, comme la sélection par le rang (les individus sont classés dans la population, et leur adaptation n'est plus fonction que du rang qu'ils occupent). On peut également envisager, lors de l'exécution de l'opérateur de croisement, de conserver parmi les deux parents et les deux enfants, les deux individus qui ont la meilleure adaptation. On effectue alors un tournoi entre les parents et les enfants à l'issue du croisement. Le danger de ce principe est la convergence prématurée de la population vers un minimum local.

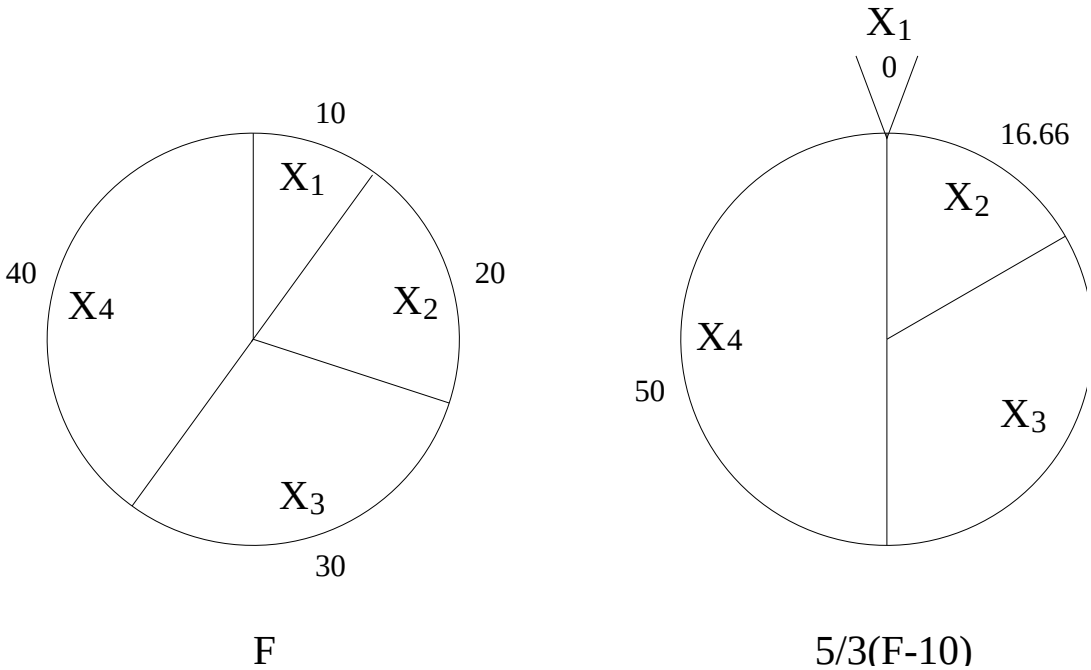

Figure 21.6 – Exemple de scaling

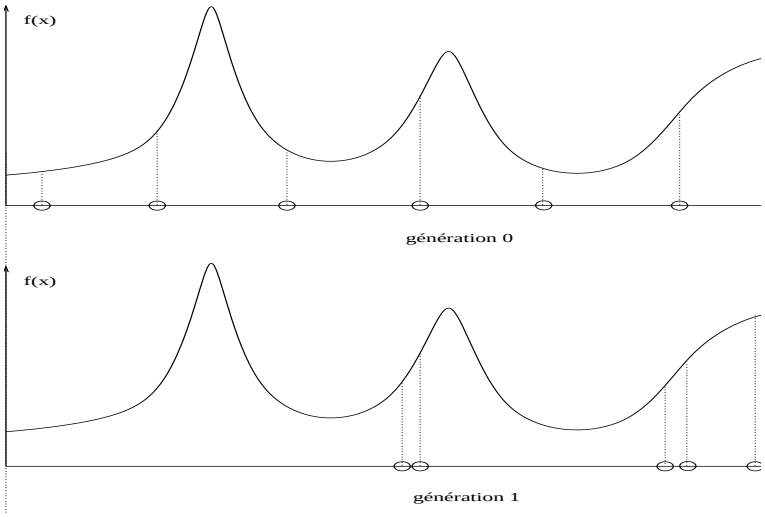

Figure 21.7 – Intérêt du sharing

21.5.5 Le sharing

Un des problèmes souvent rencontrés dans l'utilisation des algorithmes génétiques est la disparition trop rapide de certains éléments de la population en raison de la sur-domination d'un individu particulier, dont les représentants finissent par constituer la majeure partie de la population. Considérons l'exemple de la figure 21.7 : à la génération 0, la population de 8 individus est uniformément répartie sur l'intervalle de recherche. Malheureusement, les meilleurs individus ayant une plus forte probabilité de reproduction que les mauvais, on peut se retrouver, à la génération 1 avec une population répartie autour de minima locaux. Le but du sharing est de contrecarrer ce phénomène.

De plus, un des avantages des algorithmes génétiques par rapport aux algorithmes de type recuit simulé est de présenter plusieurs solutions viables, et non une seule, au terme d'une exécution du programme. Il faut donc éviter l'élimination de tous les individus au bénéfice d'un seul, même s'il est mieux adapté.

La technique dite de *sharing* (que l'on pourrait traduire par *partage* ou *répartition* en français) tente de faire disparaître ce problème. Le principe est simple : la fitness de chaque individu est modifiée en fonction du nombre d'individus qui l'entoure dans l'espace du problème ; plus ce nombre est grand, et plus on diminue la fitness. La formule classiquement utilisée est :

$$f_s(x_i) = \frac{f(x_i)}{\sum_{j=1}^{n} s(d(x_i, x_j))}$$

$f_s(x_i)$ est la valeur modifiée de la fitness de l'élément x_i après l'application de l'opération de sharing ; $f(x_i)$ est la fitness « traditionnelle » ; $d(x_i, x_j)$ est la distance séparant les éléments x_i et x_j ; s est une fonction « astucieusement » choisie. Cette fonction a souvent la forme triangulaire représentée sur la figure 21.8.

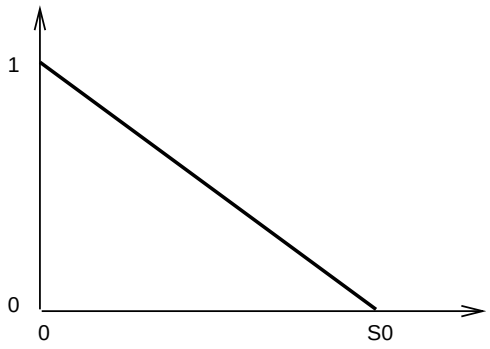

Figure 21.8 – Fonction de sharing

On voit que tout élément situé à une distance supérieure à s_0 ne modifie en rien la fitness, mais qu'en revanche, plus il y a d'éléments proches et plus la fitness se trouve diminuée. Cette technique permet ainsi de pénaliser les zones trop peuplées, contenant trop de représentants d'une même solution, ou d'une solution proche ; elle favorise en revanche les représentants uniques ou peu nombreux d'une solution.

Le sharing nécessite la définition d'une distance sur l'espace de recherche et pénalise le temps d'exécution de l'algorithme car si la population est de taille n, le temps de calcul des distances entre les paires d'individus est proportionnel à n^2. Il existe des méthodes approchées (Yin and Germay 1993) ou permettant de ramener le temps de calcul proportionnel à $n\,log(n)$.

L'utilisation du sharing sur l'exemple de résolution de conflit permet d'une part d'améliorer la solution obtenue (figure 21.9) et d'autre part d'obtenir plusieurs solutions différentes.

21.5.6 Fonctions partiellement séparables : croisement adapté

Les algorithmes génétiques sont très souvent utilisés pour des problèmes de grande taille où l'on cherche à minimiser un critère additif, somme de critères faisant chacun intervenir une partie des variables. Ce type de fonction n'est pas marginal, beaucoup de problèmes pratiques faisant intervenir un grand nombre de variables sont partiellement séparables. Nous verrons à la fin de cette section que notre problème de résolution de conflits aériens en fait partie.

L'idéal, pour résoudre un problème à plusieurs variables est de pouvoir séparer le problème en sous-problèmes indépendants. La solution globale est reconstituée à partir des solutions de chacun des sous-problèmes. Considérons l'exemple de la fonction définie sur $\Re^n$ $(x_1, .., x_n) \rightarrow \sum_1^n x_i^2$. Cette fonction est complètement séparable, elle est la somme de n critères positifs $x_i \rightarrow x_i^2$ que l'on peut minimiser indépendamment pour trouver la solution globale. Supposons que l'on n'ait pas profité de cette propriété et considérons l'opérateur de croisement simple d'un AG qui, à partir de deux parents $(x_1, .., x_n)$ et $(y_1, .., y_n)$ génère les enfants $(z_1, .., z_n)$ et $(\bar{z}_1, .., \bar{z}_n)$ avec $(z_i, \bar{z}_i) = (x_i, y_i)$ ou (y_i, x_i) . Or le meilleur enfant que l'on peut produire est obtenu en choisissant pour z_i la variable x_i ou y_i qui minimise le critère $t \rightarrow t^2$.

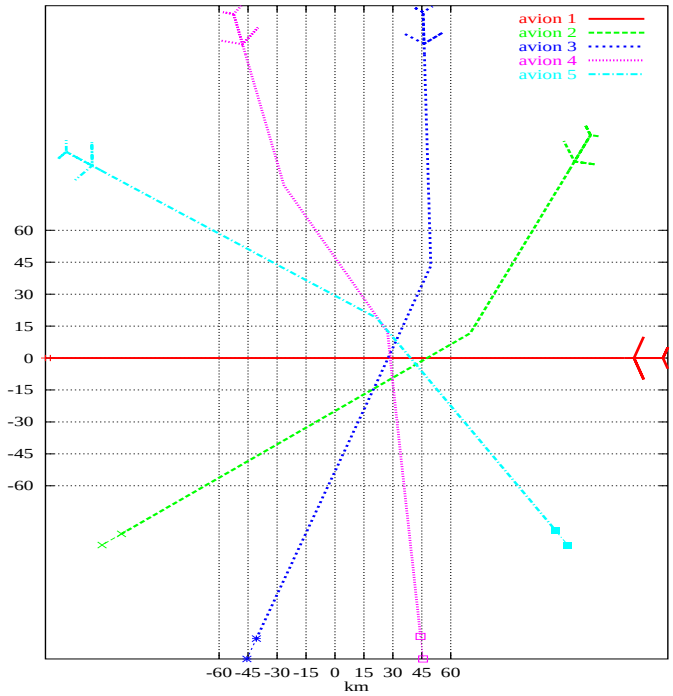

Figure 21.9 – Conflit à 5 avions avec sharing

On peut ainsi largement améliorer la convergence d'un algorithme génétique en utilisant un opérateur de croisement qui tient compte de la séparabilité du critère à optimiser pour fabriquer les enfants à partir de 2 parents.

Par la suite, on appellera fonction partiellement séparable toute fonction positive F de n variables x_1, x_2, .., x_n pouvant s'écrire comme la somme de m fonctions positives F_i, chacune ne faisant intervenir que n_i variables ($n_i < n$).

$$F(x_1, x_2, .., x_n) = \sum_{i=1}^{m} F_i(x_{j_1}, x_{j_2}, .., x_{j_{n_i}})$$

On définit l'adaptation ou *fitness locale* $G_k(x_1, x_2, .., x_n)$ pour la variable x_k comme suit :

$$G_k(x_1, x_2, .., x_n) = \sum_{i \in S_k} \frac{F_i(x_{j_1}, x_{j_2}, .., x_{j_{n_i}})}{n_i}$$

où S_k est l'ensemble des i tels que x_k est une variable de F_i et n_i le nombre de variables de F_i.

Intuitivement, l'adaptation locale d'une variable isole sa contribution à l'adaptation globale. On a de plus la propriété suivante :

$$\sum_{k=1}^{n} G_k(x_1, x_2, .., x_n) = F(x_1, x_2, .., x_n)$$

L'opérateur de croisement fabrique un enfant $z_1, .., z_n$ à partir de deux parents comme suit :

- si $G_k(x_1, .., x_n) < G_k(y_1, ..y_n) - \Delta$, alors $z_i = x_i$;
- si $G_k(x_1, .., x_n) > G_k(y_1, .., y_n) + \Delta$, alors $z_i = y_i$;
- si $|G_k(x_1, .., x_n) - G_k(y_1, .., y_n)| \leq \Delta$, alors $z_i = x_i$ ou y_i avec une probabilité $\frac{1}{2}$ ou peut être une combinaison linéaire des deux variables s'il s'agit de variables réelles.

Le paramètre Δ rend plus où moins aléatoire la construction de l'enfant.

Ce principe s'applique très bien au problème de résolution de conflits : en effet, on cherche d'abord à minimiser C somme de critères faisant intervenir des paires d'avions, puis on cherche à minimiser la somme des allongements, critères ne faisant intervenir qu'un avion chacun. Le croisement adapté permet, pour le problème de résolution de conflit, de résoudre des configurations atteignant une vingtaine d'avions, comme le montre la figure 21.10.

21.5.7 Structures de données complexes

Les algorithmes génétiques peuvent optimiser n'importe quel type de structure de données dès lors que l'on est capable de définir une fonction d'évaluation et des opérateurs de recombinaison. On peut ainsi, comme le décrit l'exemple 369 optimiser les poids d'un réseau de neurone, voire même sa structure. Pour un problème d'optimisation opérant sur des vecteurs réels, on peut définir comme élément de population non plus le vecteur lui-même, mais un simplexe de vecteurs (une méthode locale permettant d'optimiser le simplexe à chaque génération (Durand and Alliot 1999)). On peut également envisager des éléments de population diploïdes.

Diploïdie et dominance

Il était inévitable que l'on tente d'utiliser pour les algorithmes génétiques des chromosomes diploïdes en lieu et place des chromosomes haploïdes utilisés jusqu'ici : les organismes vivants les plus évolués ont des chromosomes diploïdes[12]. Dans ce cas, le décodage du génotype au phénotype se fait en introduisant la notion de dominance d'un gène sur un autre, comme en génétique classique (voir (Goldberg 1989c)).

Les diverses expériences faites sur la diploïdie tendent à montrer que cette technique n'est efficace que lorsque la fonction fitness varie en fonction du nombre de générations de l'algorithme génétique *i.e.*, f est de la forme $f(C, n)$ (où n est le numéro de génération de l'algorithme génétique), au lieu d'être simplement de la forme $f(C)$. Ceci semble être en accord avec la génétique qui pense qu'un des facteurs positifs de la diploïdie est

12 Un chromosome haploïde est un chromosome à un seul brin (une seule chaîne de bits), un chromosome diploïde est un chromosome à deux brins (deux chaînes de bits).

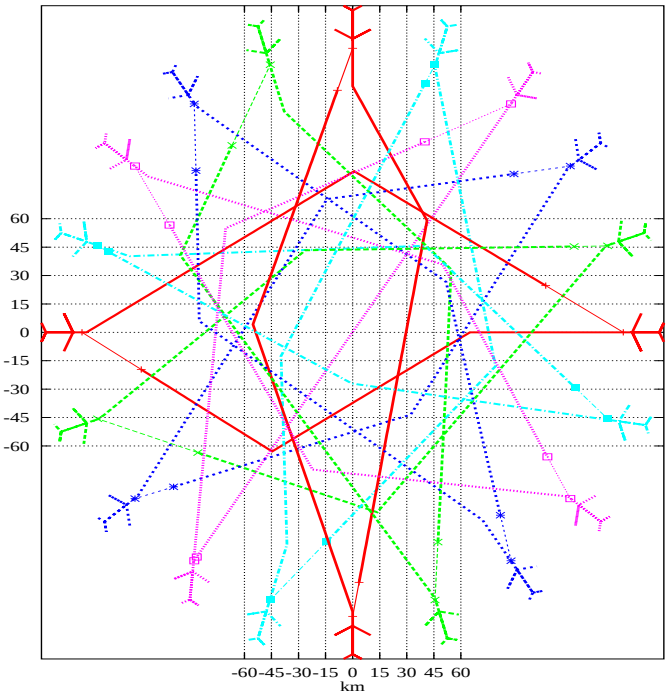

Figure 21.10 – Conflit à 20 avions

sa capacité à mémoriser (par l'intermédiaire de gènes récessifs) des caractéristiques qui ont pu se révéler utiles dans le passé, ne le sont plus aujourd'hui, et donc doivent disparaître du phénotype ; cependant, si une évolution de l'environnement venait à rendre ces caractéristiques utiles à nouveau, les gènes récessifs les codant étant disponibles, elles réapparaîtraient sur le champ.

La diploïdie est, pour l'instant, peu utilisée pour la programmation des algorithmes génétiques.

Algorithmes génétiques et réseaux de neurones

Nous allons présenter dans cette section une intéressante combinaison reliant les algorithmes génétiques aux réseaux de neurones.

Considérons le problème suivant : nous disposons dans un espace clos de six mobiles. Chacun de ses mobiles à une origine et une destination. Le but de chacun de ces engins est de rejoindre sa destination en suivant le plus court chemin. Cependant, ils ne doivent à aucun moment se trouver à une distance d'un autre mobile inférieure à une norme de séparation D, et ce durant l'intégralité de leur déplacement ; il s'agit donc de construire une trajectoire optimale au sens de la distance tout en respectant des contraintes de séparation.

Pour chaque mobile i, les paramètres en entrée sont les distances ρ_{ij} aux autres mobiles, les gisements de ces mobiles θ_{ij} ainsi que la distance ρ_i et le gisement θ_i de sa

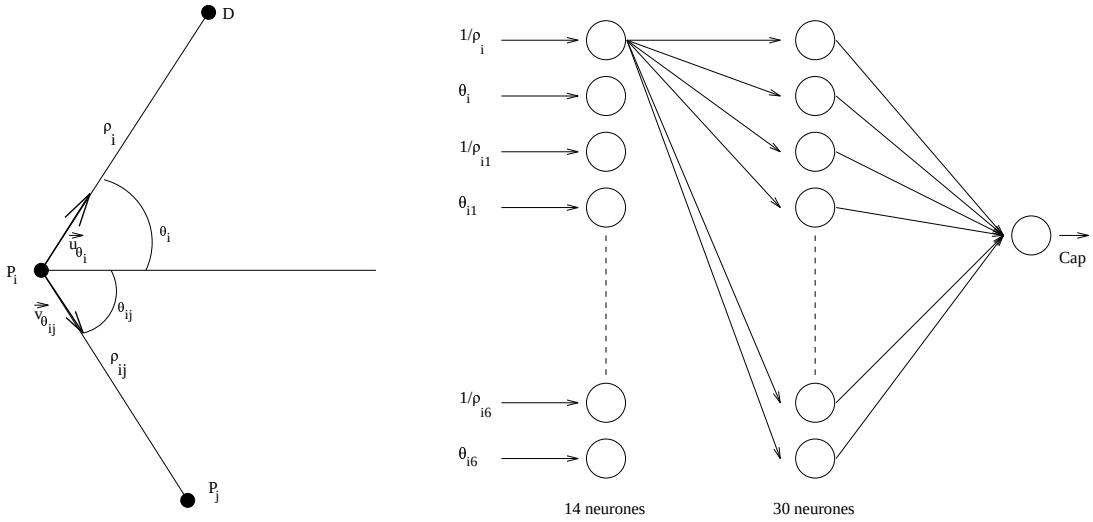

Figure 21.11 – Contrôle de mobiles par réseau de neurones

destination. En sortie, le programme doit fournir un cap à suivre α_i pour le pas de temps suivant (on suppose que les mobiles avancent par pas de temps discret, à une vitesse unique et fixée).

On peut donc construire un réseau de neurones qui aurait pour entrées les différents paramètres et fournirait en sortie le cap du mobile (voir figure 21.11). Le problème est de réaliser l'apprentissage du réseau. On ne dispose en effet d'aucune base d'exemple permettant de faire de l'apprentissage supervisé, car il n'existe pas de solution mathématique générale à ce problème.

On peut alors utiliser la technique des algorithmes génétiques pour calculer les poids du réseau de neurones. On commence par générer un ensemble de poids pour notre réseau (un chromosome contient en fait la matrice des poids). Puis, on fait se déplacer nos mobiles, sur un exemple ou un ensemble d'exemple, avec les poids de chacun des chromosomes. La *fitness* de chaque chromosome correspond à la qualité de la trajectoire générée. Le plus court chemin est bien sur l'optimal, mais il faut éviter les autres mobiles. Toute collision pénalise la fitness, soit en l'annulant, soit en la divisant par un coefficient donné.

Les opérateurs de croisement sont du type barycentrique. On tire aléatoirement deux coefficients dans deux réseaux et on réalise une combinaison linéaire pour obtenir les nouveaux coefficients. La mutation consiste à ajouter un bruit blanc sur un coefficient tiré au hasard.

On peut ainsi construire un réseau qui parviendra à réaliser les deux objectifs fixés : de bonnes trajectoires en terme de distance et un évitement des autres mobiles[13].

La combinaison des algorithmes génétiques et des réseaux de neurones ouvre une voie particulièrement intéressante pour les problèmes où l'on ne dispose pas de base de données d'exemples permettant de faire de l'apprentissage supervisé.

13 Ce travail s'inspire d'un travail similaire (Schoenauer *et al.* 1993).

21.5.8 Recherche multi-objectifs

Il est rare, pour un problème réel, que l'on n'ait qu'un seul critère à optimiser ; bien souvent, on se contente de rechercher un compromis entre plusieurs critères (par exemple une combinaison linéaire entre les critères). Or les algorithmes génétiques font évoluer une population d'individus, et on peut être tenté de faire converger cette population non plus vers un point optimum, mais vers la surface de Pareto des différents critères.

Dominance au sens de Pareto

La notion de dominance au sens de Pareto se définit comme suit : on dit que le point A domine le point B si, $\forall i, f_i(A) \geq f_i(B)$ (avec au moins une inégalité stricte), où les f_i représentent les critères à maximiser. L'ensemble des points qui ne sont dominés par aucun autre point forme la surface de Pareto. Tout point de la surface de Pareto est "optimal", dans la mesure où on ne peut améliorer la valeur d'un critère pour ce point sans diminuer la valeur d'au moins un autre critère.

On peut construire un algorithme génétique permettant de trouver la surface de Pareto d'un problème multi-critères.

Modification de l'opérateur de sélection

La technique employée dérive directement des travaux de Jeffrey Horn et Nicholas Nafpliotis ((Horn and Nafpliotis 1993)). Au lieu de travailler sur une fonction d'évaluation scalaire, on a désormais une fonction d'évaluation vectorielle (chaque critère est une variable du vecteur). En conséquence, le principal changement induit concerne le processus de sélection[14].

On introduit une variante de la notion de dominance que l'on définit ainsi : l'élément E_i domine E_j si le nombre des valeurs contenues dans son vecteur d'adaptation qui sont supérieures aux valeurs correspondantes dans E_j dépasse un certain seuil. La technique proposée pour effectuer la sélection est alors simple : on tire deux individus au hasard ainsi qu'une sous-population[15] à laquelle ils n'appartiennent pas et qui va servir à les comparer.

Trois cas se présentent alors :
- si le premier élément domine tous ceux de la sous-population, et que ce n'est pas le cas pour le second, alors le premier sera sélectionné.
- inversement, si seul le second élément domine l'ensemble de la sous-population, alors c'est lui qui sera conservé.
- il reste deux possibilités : soit les deux sont dominés, soit les deux dominent. On ne peut se prononcer sans ajouter un autre test, c'est pourquoi dans ce cas il est fait usage d'un nouveau type de sharing qui opère sur l'espace objectif.

Le sharing va conduire à sélectionner celui des deux individus qui a le moins de voisins proches, autrement dit, on élimine celui qui se situe dans une zone d'agrégation pour conserver celui qui est dans une région moins dense.

---

14 En ce qui concerne les autres opérateurs de base, à savoir le croisement et la mutation, il n'y a aucune modification, car ceux-ci travaillent dans l'espace d'état et non dans l'espace résultat.

15 La sous-population tirée aura une taille proportionnelle à celle de la population de départ (généralement autour de 10%). Seule une partie des individus est ainsi utilisée, ce qui permet de réduire le temps de calcul.

Le terme *voisin proche* n'a pas de signification précise mais il est possible de définir un voisinage (aussi appelé niche) correct en se servant de la distance de Holder :

$$d_H(E_i, E_j) = \left(\sum_{k=1}^{n} |f_i^k - f_j^k|^p \right)^{\frac{1}{p}}$$

f_i^k désignant la k-ième composante du vecteur adaptation de l'élément i. Le paramètre p permet de faire varier la forme et la taille du voisinage. A l'intérieur des voisinages ainsi définis dans l'espace objectif, il suffit de compter le nombre d'individus pour favoriser les zones les moins denses, et de cette façon, maintenir la diversité de la population.

21.5.9 Algorithmes génétiques et coévolution

Pour certains problèmes d'optimisation, la fonction fitness résulte d'une évaluation sur un certain nombre de cas tests. Or bien souvent, il est difficile de choisir une base de cas tests représentative de tous les cas possibles. Imaginons par exemple que l'on veuille approcher la courbe que nous fournit une boîte noire par un polynôme de degré n et que l'on veuille déterminer les coefficients du polynôme en utilisant un algorithme génétique. On peut sélectionner un certain nombre de points de la courbe qui permettront, pour un polynôme donné de mesurer la qualité du polynôme retenu. Rien ne garantit toutefois qu'une autre sélection de points ne donne pas une évaluation totalement différente.

Le principe de la coévolution est de faire évoluer deux populations différentes en parallèle. Dans le cas présent, la première population est une population d'individus codant les coefficients d'un polynôme, la deuxième population est une population de points de la courbe servant à tester la première population.

Il n'existe pas d'algorithme de coévolution *canonique*. Différents travaux ayant utilisé le principe de coévolution permettent de donner une image plus concrète de ce que peut être la coévolution :

– W. D. Hillis a été un des premiers à utiliser des techniques de coévolution du type *proie-prédateur*. Ce nom provient du fait qu'il s'agit de faire évoluer simultanément, mais de manière concurrente, deux populations. Hillis emploie le mot de *parasites* pour désigner les éléments de la population concurrente, qui en exploitant les failles éventuelles des éléments de la première population, les poussent à s'améliorer.
  Dans un article de 1992, *Co-evolving Parasites Improve Simulated Evolution as an Optimization Procedure* (Hillis 1992), Hillis présente une méthode dont le but est de générer des réseaux de tri, parvenant à trier un nombre n donné de nombres au moyen du plus petit nombre possible de comparaisons successives. Les éléments de la première population sont des réseaux de tri, qui évoluent au moyen d'un algorithme particulier. Nous développerons les caractéristiques de cet algorithme qui nous semblent jouer un rôle important dans l'utilisation par Hillis de la coévolution. Les *parasites* sont des ensembles de *cas tests*, c'est-à-dire des séquences d'entiers (de 0 et de 1 puisque un réseau de tri qui trie correctement toute séquence de 0 et de 1 de taille n trie correctement toute séquences de nombres de taille n). Les réseaux ont une adaptation d'autant plus grande qu'ils parviennent à ordonner correctement une plus grande proportion de ces cas tests La coévolution

consiste dans ce cadre a faire évoluer, en même temps que la population de réseaux, une population de *parasites*.
Hillis obtient grâce à la coévolution de meilleurs résultats que ceux qu'il avait obtenus au moyen de l'évolution simple des réseaux, tant en termes d'efficacité des réseaux pour un plus petit nombre de comparaisons, qu'en terme de rapidité d'obtention de ces réseaux. Il voit à ce résultats deux explications. D'abord, les réseaux imparfaits (qui ne trient pas correctement toutes les séquences de n entiers), qui pourraient constituer des optima locaux sont la proie de parasites adaptés à leurs faiblesses qui se développent dans leur voisinage, et ainsi ne sont pas viables (on voit ici le rôle de la localisation des éléments des deux populations sur une grille). De plus, les cas tests les plus difficiles dominent rapidement dans la population des parasites, ce qui évite des évaluations inutiles de réseaux sur des cas tests trop faciles.

– Jan Paredis (Paredis 1994; Paredis 1996b; Paredis 1996a) utilise aussi des méthodes de coévolution du type *prédateur-proie*. Il utilise conjointement un algorithme de coévolution (CGA : Co-evolutionary Genetic Algorithm), et une technique d'apprentissage, qu'il appelle *Life Time Fitness Evaluation* (LTFE). Il donne dans *coevolutionary computation* (Paredis 1996a) une présentation générale de son algorithme CGA tel qu'il peut être appliqué à ce qu'il appelle des *test-solution problems*. c'est-à-dire des problèmes dans lesquels on cherche à obtenir des solutions générales à appliquer à un grand nombre de cas différents, et pour lesquels on peut disposer d'un (grand) nombre de cas tests. La recherche de réseaux de tri décrite dans l'article de Hillis est un cas particulier de problème solution-test, et Jan Paredis développe dans les différents articles cités ici l'application de son algorithme à différents problèmes tels que la recherche de réseaux neuronaux de classification (classification neural networks) (Paredis 1996a), ou des problèmes de satisfaction de contraintes (CSP) (Paredis 1994; Paredis 1996a).

21.6 Stratégies d'évolution

Les stratégies d'évolution sont en fait généralement désignées, même dans les milieux francophones, par leur nom anglais de *Evolution Strategies*, ou par le sigle *ES*. Elles ont été introduites par Schwefel, à Berlin en 1965. à l'origine, un individu unique produisait, au moyen d'une mutation, un descendant unique, une sélection s'opérait entre eux deux.

Dès leurs origines, une grande originalité des stratégies d'évolution par rapport aux algorithmes génétiques a résidé dans leur utilisation du codage réel (voir section 21.5.1) et de la mutation gaussienne (section 21.5.1).

Elles ont été étudiées et développées par Rechenberg, à qui on doit la règle dite du un cinquième : le taux de mutation est modifié au cours du déroulement de l'algorithme de manière à atteindre un taux de succès des mutation de un cinquième, c'est-à-dire que dans un cas sur cinq en moyenne, le produit d'une mutation (le *fils*) a une meilleure *fitness* que l'élément à partir duquel il a été obtenu (le *père*). Des techniques similaires ont pu ensuite être reprises dans le cadre d'autres algorithmes évolutionnaires, de même que la *self adaptation*, introduite par Schwefel (voir section 21.5.1).

Dans les années 70 ont été développées 2 versions des stratégies d'évolution, les $(\mu + \lambda)$-ES et les (μ, λ)-ES, dues à Schwefel, dont on trouvera une description dans (Bäck and Schwefel 1993). Dans les deux cas, μ individus donnent, au moyen de mutations (et éventuellement d'opérateurs de recombinaisons, tels que le croisement décrit dans la section 21.5.1), λ nouveaux individus. Dans le cadre des (μ, λ)-ES, $\lambda > \mu$, et la sélection retient μ individus sur les λ qui ont été créés. Elle retient, dans le cadre des $(\mu + \lambda)$-ES, μ individus sur l'ensemble des μ *parents* et des λ *enfants*. La sélection est déterministe : les meilleurs λ individus (en termes de *fitness*) sont conservés, les λ moins bons sont éliminés.

Les stratégies d'évolution, en plus de former un groupe d'algorithmes très intéressant en eux mêmes, ont eu une influence positive sur développement des algorithmes génétiques, notamment par la mise au point de nouveaux opérateurs de mutation, et par l'expérience du codage réel qu'elles ont permis d'acquérir.

21.7 Programmation évolutionnaire

Le principe de la programmation évolutionnaire, ou Evolutionnary Programming a d'abord été présenté par L. J. Fogel (Fogel *et al.* 1966) en 1966.

Ces algorithmes ont été grandement développés (notamment par le passage au codage réel) et popularisés par D. B. Fogel (Fogel *et al.* 1992).

Leurs principales caractéristiques, par rapport aux autres algorithmes évolutionnaires, étaient, à l'origine, le fait qu'ils n'utilisaient pas de croisement, et surtout l'utilisation d'un mode de sélection originale, la sélection par tournoi.

Nous donnons ici une description simple d'un algorithme de programmation évolutionnaire (cette famille d'algorithmes s'est rapidement enrichie et diversifiée). μ individus en donnent μ par mutation. Si le codage utilisé est un codage réel, cette mutation est gaussienne, et son écart-type peut, au moment de la mutation d'un individu, être calculé à partir de la *fitness* de cet individu.

Notons $\mathcal{P}(n)$ la population des μ individus présent à la génération n, et $\mathcal{P}'(n)$ les μ individus obtenus à partir d'eux par mutation. On va d'abord attribuer une note à chacun des 2 μ individus de $\mathcal{P}(n) \cup \mathcal{P}'(n)$. Pour chaque individu e de $\mathcal{P}(n) \cup \mathcal{P}'(n)$, on tire aléatoirement q individus dans $\mathcal{P}(n) \cup \mathcal{P}'(n)$. La note de e est alors égale au nombre d'individus (parmi ces q) dont la fitness est inférieure à celle de e.

La sélection retient pour former la nouvelle population $\mathcal{P}(n+1)$, les μ éléments ayant obtenu les meilleures notes.

Les performances de représentants de ces deux familles d'algorithmes (ES et EP), et d'un algorithme génétique (utilisant un codage binaire) sont comparées sur des problèmes d'optimisation de fonctions test de $\Re^n$ dans $\Re$, dans (Bäck and Schwefel 1993).

21.8 La programmation génétique

La programmation génétique est une technique récente qui cherche à appliquer les techniques génétiques sur des ensembles de programmes et non sur des données. Nous

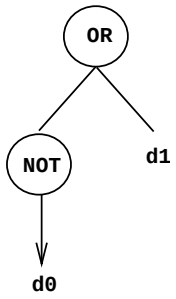

Figure 21.12 – Fonction LISP représentée par un arbre

ne décrirons que très brièvement la programmation génétique, le lecteur intéressé devra se reporter à (Koza 1992) : une présentation exhaustive, accompagnée de nombreux exemples, des techniques de programmation génétique.

Prenons un exemple simple : supposons que nous souhaitions construire une fonction LISP à partir des fonctions logiques classiques (`AND`, `OR`, `NOT`) qui implante l'implication. Il est clair que dans ce cas, la fonction s'écrit simplement :

```
(define (implication d0 d1) (OR (NOT d0) d1))
```

La programmation génétique permet de générer automatiquement une telle fonction à partir de la table de vérité de l'implication. Le fonctionnement est le suivant :

1. on commence par générer une population originelle comprenant, par exemple, 200 programmes. Un programme est représenté par un arbre ne contenant que les nœuds `AND`, `OR` et `NOT`, et dont les terminaux sont `d0` ou `d1`. On peut voir sur la figure 21.12 la représentation de la fonction LISP implantant l'implication.
2. on calcule ensuite la fitness de chacune des fonctions. Dans le cas présent, la fitness est simplement le nombre de fois où la fonction prédit correctement la sortie pour chacune des entrées possibles ;
3. les programmes se reproduisent alors en fonction de leur fitness, comme pour les algorithmes génétiques classiques ;
4. on effectue ensuite l'opération de croisement. On tire au hasard deux fonctions dans la population et on tire sur chacune de ces deux fonctions un site de croisement au hasard : un site de croisement est une branche, quelconque, de l'arbre représentant la fonction. On remplace alors dans le premier arbre la partie coupée par la partie coupée du second arbre et réciproquement. On peut voir un exemple de croisement sur la figure 21.13 ;
5. on effectue ensuite les mutations. Il s'agit de sélectionner au hasard une fonction, puis un point de mutation dans cette fonction, de couper l'ensemble du sous-arbre situé sous ce point et de le remplacer par un sous-arbre généré aléatoirement ;
6. on peut alors reprendre à l'étape (2) la génération suivante.

Un des problèmes principaux de la programmation génétique est le choix des opérateurs à introduire dans la population initiale. Dans notre cas, le choix de `NOT`, `AND` et `OR` était relativement simple à faire. Dans bien d'autres cas, il n'en est pas de même, et le choix initial des opérateurs conditionne bien souvent la réussite ou l'échec du mécanisme

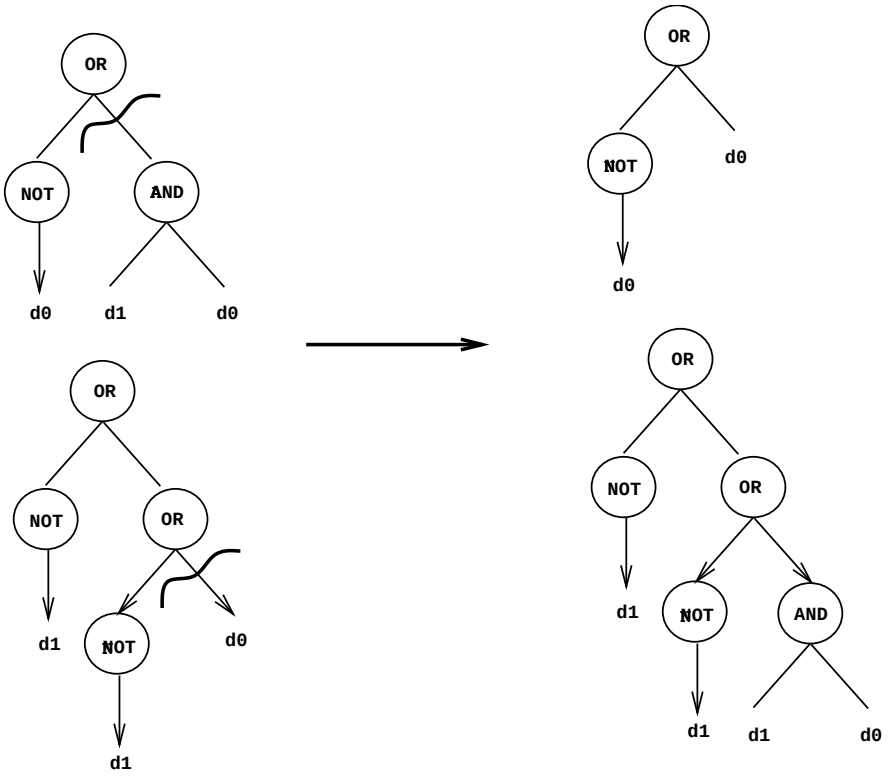

Figure 21.13 – Croisement de deux programmes

de programmation génétique. Comme dans bien d'autres cas, le savoir-faire de la personne qui met la technique en œuvre conditionne le résultat bien plus sûrement que la technique elle-même.

21.9 Conclusion

Au même titre que les réseaux de neurones, bien que dans une optique différente, les algorithmes génétiques représentent une approche originale et intéressante. Là aussi, il faut se garder de se laisser prendre aux phénomènes de mode et songer à comparer les algorithmes génétiques à des méthodes plus classiques d'optimisation (recuit simulé, branch and bound, etc.) On peut considérer qu'ils sont passés du stade de la recherche pure à celui de la recherche appliquée.