Contribution à l'amélioration des techniques de la programmation génétique

Oussama El Gerari

THÈSE

présentée et soutenue publiquement le 08 Décembre 2011
en vue de l'obtention de grade de

Docteur de l'Université du Littoral Côte d'Opale

Discipline : Informatique

par

Oussama EL GERARI

Contribution à l'Amélioration des Techniques de la Programmation Génétique

Composition du jury

Président.	Henri BASSON Professeur, Université du Littoral Côte d'Opale.
Rapporteurs	Adnan YASSINE Professeur, Université du Havre. Mohamed SLIMANE Professeur, Université de Tours.
Directeur	Cyril FONLUPT Professeur, Université du Littoral Côte d'Opale.

Laboratoire d'Informatique, Signal et Image de la Côte d'Opale -EA 4491- Équipe MODEL
Maison de la Recherche Blaise Pascal- Centre Universitaire de la Mi-Voix
50 rue Ferdinand Buisson – B.P. 719, 62228 Calais Cedex, France

À MA MÈRE. À MON PÈRE..

Remerciements

Cette thèse a été effectuée au sein du laboratoire LIL (Laboratoire d'informatique du Littoral), avant qu'il ne s'agrandisse suite à une fusion pour devenir LISIC (Laboratoire d'Informatique, Signal et Image de la Côte d'Opale), à l'université du Littoral côte d'Opale.

Mes Premières pensées vont à mon directeur de thèse, le Professeur Cyril FONLUPT, qu'il trouve ici l'expression de ma profonde gratitude pour m'avoir accueilli, dirigé et surtout soutenu tout au long de cette thèse.

Je tiens à remercier mes rapporteurs, Mr. Mohamed SLIMANE et Mr. Adnane YASSINE, pour leur patience en révisant mon travail de thèse.

Un grand merci aux membres du Laboratoire LISIC pour leur accueil et leur sympathie. Merci au directeur du Laboratoire le Professeur Christophe Renaud pour son soutien. Merci aux membres de mon équipe MODEL (Multi-modélisation et évolution des logiciels) et spécialement à Denis ROBILLARD et Virginie MARION-POTY pour l'aide et la chaleur humaine. Merci à Dominique VERHAGHE, Mourad BOUNEFFA, Henri BASSON. Et mes amis de la salle B125, Hanaa MAZAYD, Adeel AHMAD, Oussama-Mohammed KHERBOUCHE.

Merci à mes Amis, Aïmène EL GERARI, Taoufik EL HAJLI, Abdelhafid KASMI et tant d'autres dans mon cœur, pour les témoignages d'amitié et le soutien inconditionnel durant ces années de thèse.

Merci à ma Famille, et surtout à mes **PARENTS,** qui, sans eux, je n'aurai pas eu courage et ambition.

Table des matières

Introduction générale

Contexte du travail

Le rêve de construire une machine intelligente est ancestrale, mais c'est grâce à l'informatique que ce rêve a commencé à prendre forme. Plusieurs domaines participent à cette quête. Une des pièces du Graal est la programmation automatique dont le principe consiste à générer automatiquement de nouveaux programmes. Si une machine est capable de générer des programmes de manière automatique avec une aide minimale de l'humain, un petit pas aura été franchi vers une machine « intelligente ».

Parmi les algorithmes qui constituent la famille de la programmation automatique, on s'intéresse à la Programmation Génétique (PG) [Koz 1992]. Comme tous les algorithmes évolutionnaires, cette méthode s'appuie sur la théorie de l'évolution de Darwin. En effet, Koza s'est inspiré des algorithmes génétiques de Holland [Hol 1975] et a utilisé les arbres comme forme de représentation, ce qui permet de générer des programmes et permet de les exécuter et de leur attribuer une valeur qui donne des informations sur leur qualité.

La programmation génétique et les algorithmes évolutionnaires sont désormais établis en tant qu'outils performants et comme algorithmes d'optimisation efficaces parmi les techniques d'intelligence artificielle et humainement compétitive, car leurs résultats sont comparables à des inventions brevetées, sinon meilleures. Parmi les résultats les plus reconnus, citons la création d'antennes satellite pour équiper une des navettes spatiales de la NASA [Loh 2004].

L'espace de recherche des programmes possibles est déterminé par la taille de la grammaire sous-jacente qui constitue le jeu d'instructions qui sera utilisé. Si la taille de la grammaire est trop restreinte, l'espace de recherche sera petit et les solutions trouvées risquent de ne pas posséder une qualité satisfaisante. Inversement, si la taille de la grammaire est trop importante, la recherche risque d'être coûteuse en terme temps d'exécution et en ressources.

Motivations et objectifs

Dans le cadre général de cette thèse, nous nous intéressons à améliorer les techniques de programmation génétique, en particulier nous avons essayé d'améliorer la performance de la PG en cas d'utilisation de grammaire riche, où l'ensemble de terminaux contient plus que nécessaire pour représenter des solutions optimales. Pour cela, nous avons présenté le problème de la sélection d'attributs en rappelant les

principales approches et nous avons utilisé la technique de mesure de poids des terminaux pour affiner la sélection d'attributs.

En second lieu, nous présentons des travaux sur un autre algorithme qui s'inspire de la boucle évolutionnaire : l'évolution différentielle (ED), et nous étudions la performance de cette technique sur la branche de la programmation génétique linéaire.

Organisation de la thèse

- Dans le premier chapitre, nous présenterons les algorithmes évolutionnaires les plus importants et nous présenterons de manière plus détaillée la programmation génétique ;
- Au deuxième chapitre, nous présenterons le contexte général du problème de sélection d'attributs ;
- Au troisième chapitre, nous présenterons les résultats des méthodes que nous avons développées pour améliorer les performances sur des problèmes de régression symbolique ;
- Au chapitre 4, nous présenterons la programmation génétique linéaire ainsi que la méthode de génération automatique de programmes que nous avons créée à partir de la technique d'évolution différentielle et nous comparons les résultats de cette dernière avec celle de la programmation génétique sur des problèmes de régression et le problème de la fourmi artificielle. ;
- Le dernier chapitre est une synthèse des différents travaux et présente également quelques perspectives de recherche.

1 Chapitre 1 : Introduction aux algorithmes évolutionnaires

1.1 Introduction

L'homme s'est toujours inspiré de la nature qui l'entoure pour avancer et améliorer sa vie. L'informatique a également utilisé ce paradigme et des domaines comme les réseaux de neurones et la vie artificielle ont été le fruit d'une inspiration réussie de l'évolution.

Les algorithmes évolutionnaires (Evolutionary Algorithms ou Evolutionary Computation). discipline de l'intelligence artificielle, sont des algorithmes d'optimisation stochastique simulant l'évolution naturelle. Ces algorithmes sont inspirés du paradigme de l'évolution darwinienne des espèces, telle qu'elle a été définie par Charles Darwin [Dar 1886]. Les espèces naturelles sont en compétition pour survivre et seules les plus aptes survivent à la sélection naturelle. Au cours de leur évolution, ces mêmes individus auront la possibilité de transmettre leur patrimoine génétique à la génération suivante par reproduction. L'itération de ce principe permet pendant des générations de faire apparaître dans la population des individus plus adaptés à leur environnement.

Les algorithmes évolutionnaires sont particulièrement utiles pour la résolution des problèmes où les algorithmes classiques d'optimisation, d'apprentissage ou de conception automatique sont incapables de donner des résultats satisfaisants. L'introduction du principe évolutionnaire date pas d'hier [Fog1966][Hol 1975], mais ce n'est que vers les années quatre-vingt-dix qu'on a commencé à profiter de la puissance des machines pour traiter des problèmes réels de taille importante.

Dans ce chapitre nous allons voir une présentation succincte des principaux algorithmes évolutionnaires.

1.2 Histoire des algorithmes évolutionnaires

La discipline compte quatre branches principales : les algorithmes génétiques (Genetic Algorithms[Hol 1975, Gol 89]), les stratégies d'évolution (Evolution strategies [Rech 1973]) la programmation évolutionnaire (Evolutionary Programming [Fog 1966]) et la programmation génétique Genetic Programming [Koz 1992]). D'autres branches se rapprochent des algorithmes évolutionnaires comme l'évolution différentielle[Sto 1997]] que l'on abordera dans le chapitre quatre ou encore l'évolution grammaticale [One 2003] branche de la programmation génétique. Toutes ces méthodes n'utilisent pas une solution unique, mais se basent sur un ensemble de solutions qui évolue qu'on appelle population.

1.2.1 Les algorithmes génétiques (AG)

Les AG sont les fruits des travaux de John Holland[Hol 1975]. Ils ont été nommés au début « adaptive plan » et ont été popularisés par Goldberg [Gol 89]. Les algorithmes génétiques impliquent l'évolution d'une population de vecteurs de longueur fixe composée généralement de bits, même si des versions actuelles travaillent sur les vecteurs de réels ou même sur des structures plus complexes. Les AG sont utilisés dans le but de découvrir une solution à un problème donné, sans information ou peu d'information a priori sur l'espace de recherche. Au cours de l'évolution, on utilise des opérateurs inspirés de l'évolution naturelle (la mutation qui modifie un bit du vecteur et le croisement entre deux vecteurs pour en produire un nouveau). Un critère de qualité est nécessaire pour discriminer différentes solutions, cette fonction s'appelle fitness ou fonction objective ou fonction d'adéquation.

1.2.2 Les stratégies d'évolution (SE)

Cette technique a été développée par Ingo Rechenberg et Hans Paul Schwefel [Rech 1973, Sch 1981, Sch 1965]. Les populations utilisées par les stratégies d'évolution sont représentées par des vecteurs de nombres réels de dimension fixe, qui représentent les caractéristiques d'une solution potentielle. Les SE sont souvent nommées sous la forme (μ,λ)-ES ou (μ+λ)-ES qui désigne deux différentes stratégies, où μ désigne la dimension de l'ensemble de parents pour produire un ensemble de descendants de dimension λ(>= μ). (μ,λ)-ES indique que μ vecteurs sont choisis pour former la nouvelle génération parmi les meilleurs vecteurs λ. Pour la stratégie (μ+λ)-ES, les μ vecteurs de la nouvelle génération sont sélectionnés parmi les meilleurs entre les μ parents et leurs λ descendants.

1.2.3 La programmation évolutionnaire (PE)

La programmation évolutionnaire a été introduite par Fogel [Fog 1966], et a été initialement conçue pour faire évoluer les machines à états finis et a été étendue par la suite aux problèmes d'optimisation de paramètres. À la différence des autres branches de la famille des algorithmes évolutionnaires, la PE n'utilise pas une représentation spécifique. Elle se focalise sur l'opérateur de mutation approprié à un problème spécifique. Pour résoudre un problème, on génère aléatoirement une population de taille μ, et chaque individu *i* va générer λ descendants suite à l'application de l'opérateur de mutation. Une opération de sélection naturelle permet par la suite de former une nouvelle génération de taille μ à partir des parents et des descendants.

On va par la suite étudier de manière plus détaillée la quatrième branche des algorithmes évolutionnaires : la programmation génétique.

1.2.4 La programmation génétique (PG)

John Koza [Koz 1992] a été le premier à exprimer formellement le concept de PG au début des année 1990 en s'appuyant sur les techniques des algorithmes génétiques pour faire évoluer une population de programmes sous forme arborescente (les solutions ou individus de la population sont donc des programmes). Cependant, la première utilisation d'une structure arborescente dans un algorithme génétique a été définie par Cramer en 1985[Cra 1985].

1.3 Principe général de la programmation génétique

Résoudre un problème de manière automatique a toujours été d'un intérêt central, idée abordée dès la fin des années 1940 par Alan Turing [Tun 1950].

Le domaine des *systèmes Intelligents* a toujours eu pour but de produire des systèmes ayant un comportement supposé intelligent. La PG s'est inspirée du système de l'évolution naturelle (source de l'intelligence sur terre) et a pour but de résoudre des problèmes de manière automatique ; une démarche qui nécessite de l'intelligence si la même tache est accomplie par un être humain, qui n'est autre que la définition donnée par Arthur Samuel [Sam 1983] sur le but du domaine de l'apprentissage automatique et des systèmes intelligents.

Dans le cadre de la programmation génétique, à partir d'une première population de programmes générée de manière stochastique et à l'aide d'opérateurs inspirés librement du darwinisme, la PG fait évoluer cette population de manière stochastique. Par réitération de ce processus, on espère faire converger la population vers des solutions (programmes) répondant au problème à résoudre. Le schéma (suivant 1.1) donne une idée du fonctionnement général de la programmation génétique :

1. phase d'initialisation : on génère des individus de manière aléatoire : les individus de la première population. Cette population est évaluée afin d'estimer son adéquation par rapport au problème à résoudre ;
2. sélection des meilleurs individus et application d'opérateurs génétiques (mutation, croisement, recopie,...) ;
3. calcul de la qualité des individus (fitness), puis remplacement de l'ancienne génération par la nouvelle génération d'individus générée en 2;
4. retour à l'étape 2 si le critère d'arrêt n'est pas satisfait (solution satisfaisante trouvée, ou arrêt après un critère comme le nombre maximal de générations) .

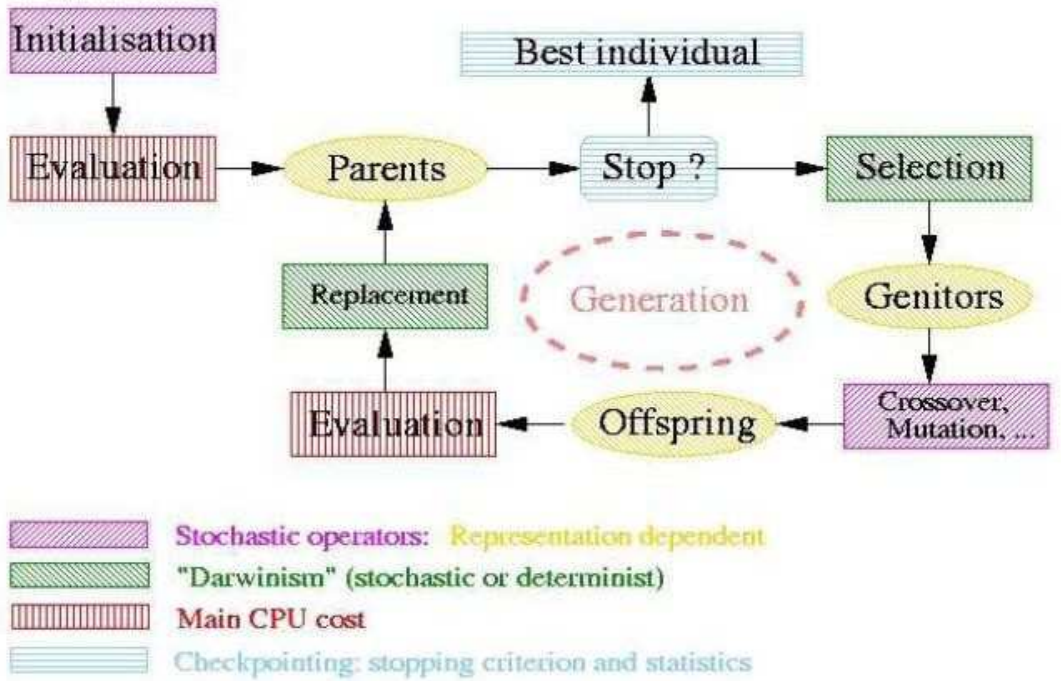

Figure 1: **Schéma d'évolution de la programmation génétique**

Description de l'algorithme général de la PG (algorithme1):

Algorithme1 : Algorithme Génétique
Initialisation de la population
répéter
Sélection des individus
Application des opérateurs génétiques sur ces individus
Évaluation de nouveaux individus
jusqu'à avoir une solution satisfaisante

1.3.1 Représentation des individus en PG

De manière grossière, en biologie, les informations portées par un gène sont appelées génotype et on appelle phénotype le caractère exprimé par ce gène. Par transposition, en PG, un programme peut se voir suivant deux angles: génotypique, la forme sur laquelle s'applique les opérateurs génétiques. et phénotypique, la forme suivant laquelle la fonction objective (fitness) sera évaluée.

La forme génotypique la plus utilisée en PG est la forme arborescente, où chaque programme est codé sous la forme d'un arbre.

Koza a utilisé cette forme pour implanter des programmes. Cette forme est la transposition directe de la forme préfixée, utilisée par exemple par le langage Lisp [Mcc 1959]. Ainsi, la formule (+ (min x (* y 2)) (+x 6)) représente le programme min(x,y*2)+(x+6).

L'intérêt de la représentation arborescente s'explique par la notion de fermeture (closure) car il est difficile de réaliser un croisement quelconque de deux programmes dans un langage haut niveau et obtenir un nouveau programme valable, contrairement au croisement de deux arbres qui donne forcement un nouvel arbre qui représente un programme valide.

Un programme aura donc la forme du graphe acyclique présenté dans la figure 2. Ce graphe est constitué de deux types de nœuds reliés par des arcs, les nœuds internes comme (min, +,*) sont des nœuds représentant les *fonctions* et les feuilles comme (x, y,2,6) représentent les *terminaux ;* ces nœuds ne peuvent en effet avoir des fils.

Les fonctions peuvent aussi bien être des fonctions arithmétiques (-, +, *, …), des fonctions booléennes, ou encore des fonctions spéciales nécessaires pour résoudre un problème particulier. En général, les terminaux sont des constantes ou des variables utilisées comme entrées pour le programme.

L'exemple présenté dans la figure 2 présente des fonctions d'arité 2 (c'est-à-dire prenant deux arguments en paramètre) mais il est évident que des fonctions d'arité quelconque peuvent être utilisées.

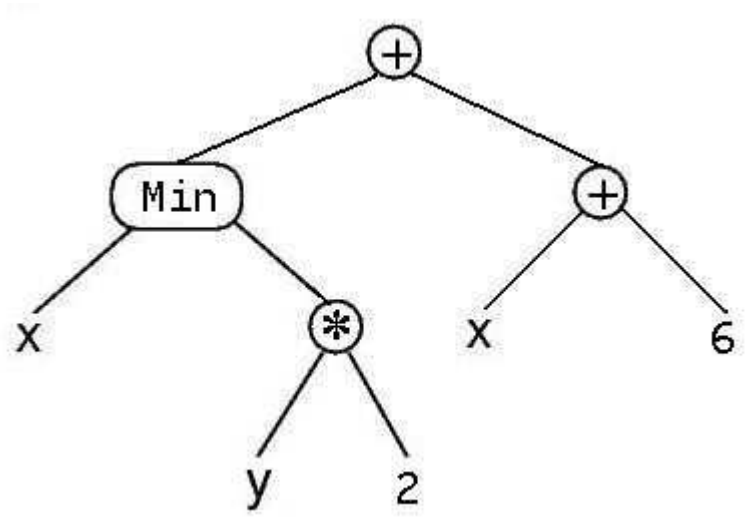

*Figure 2: Syntaxe du programme min(x,y*2)+(x+5)*

D'autres représentations existent comme celle basée sur le concept de grammaire formelle[Rya 1998, RAT 2001]. Cette représentation se base sur la représentation classique d'une grammaire {*S, N, T, P*}, S symbole de départ, N ensemble de *symboles non-terminaux,* T ensemble de *symboles terminaux,* P ensemble de *règles de production*.

La figure 3.a présente une grammaire qui peut générer un polynôme avec deux variables x,y de degré n. La figure 3.b montre un exemple d'arbre généré correspondant à la grammaire. L'utilisation de grammaire formelle permet d'imposer certaines contraintes qui peuvent être utiles pour la résolution du problème. Autre avantage de l'utilisation de grammaires est qu'elle permet une réduction considérable de l'espace de recherche de PG en imposant une forme « particulière » aux solutions.

Une représentation alternative qui a reçu une grande attention ces dernières années est la représentation linéaire qui utilise directement les instructions de langage informatique[Ban 1998][Bra 2007]. Les génotypes ou programmes sont représentés sous forme d'instructions qui manipulent directement des variables. Nordin a poursuivi cette idée en utilisant le langage machine [Nordin 1997]. En améliorant cette méthode, il a réussi à manipuler les individus en tant que code binaire et les exécuter sans passage par un interpréteur lors du calcul du *fitness*. L'avantage qui en résulte est la compacité. Il existe d'autres représentations parmi lesquelles la représentation des individus sous forme de graphes [Tell 1995, GRU 1992].

$$N = \{< E >, < O >, < V >\}$$
$$T = \{ +, *, min, x, y, \mathcal{R}\}$$

$$\left\{ \begin{array}{c} S :=< E >; \\ < E >:=< O >< E >< E > \mid < V >; \\ < O >:= + \mid * \mid min \mid x \mid y; \\ < V >:= x \mid y \mid \mathcal{R} \end{array} \right\}$$

Figure 3.a: grammaire pour représenter les programme; <E> symbole non-terminal, qui a deux forme <V> symboles terminaux ou <O><E><E>, où <O> ensemble des opérateurs

1.3.2 Initialisation de la population

La première étape de tout algorithme évolutionnaire est la création d'une population initiale. Cette population est généralement générée d'une manière aléatoire. Dans le cadre de la PG, Koza a défini deux principaux types de génération aléatoire des programmes lors de l'initialisation : les algorithmes *grow* et *full*. Les deux algorithmes ont une taille maximale de profondeur à ne pas franchir que l'on doit spécifier à l'avance.

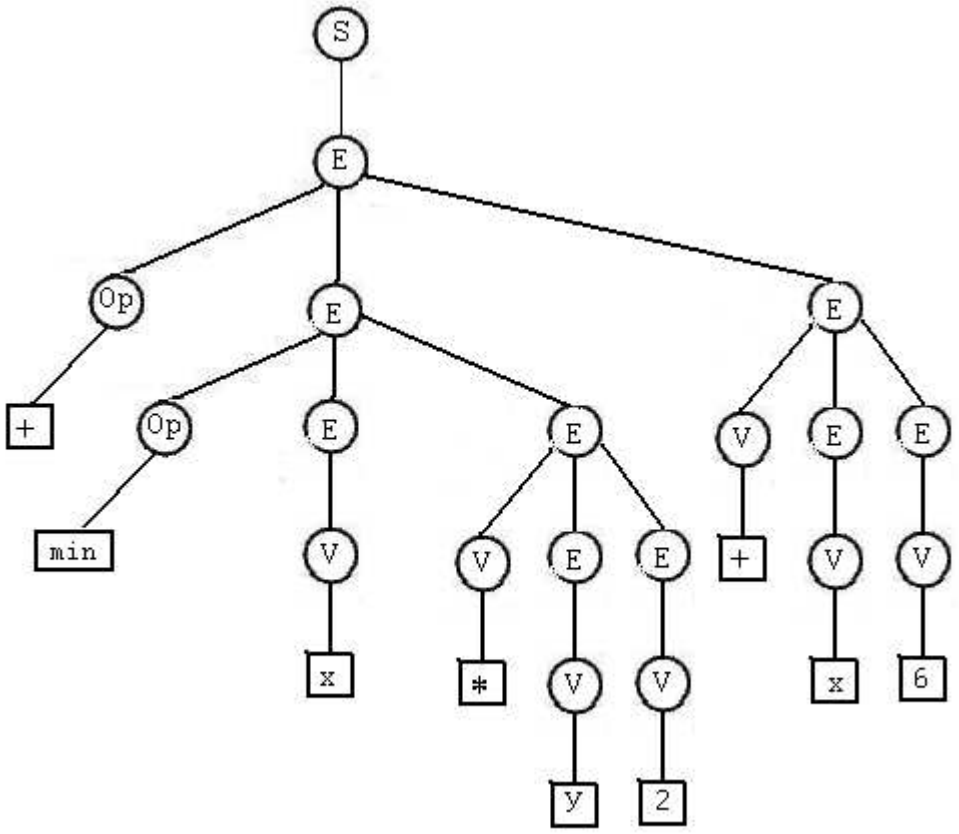

Figure 3.b: l'arbre correspondant

La méthode « *grow* » va créer des arbres de taille irrégulière et avec des formes différentes, car pour le choix de chaque nœud hormis à la profondeur maximale, on a le choix entre les ensembles terminaux et fonctions. Une fois la profondeur limite atteinte, l'algorithme ne choisira que des terminaux. La figure 4 nous montre le processus de construction d'un programme de profondeur 2, la racine dans cet exemple est la fonction « plus ». Arrivé à la profondeur maximale 2, seuls des terminaux pourront être choisis. L'algorithme 2 présente la méthode.

L'algorithme « grow » permettant de créer des arbres ayant une forme quelconque et une profondeur fixée bornée.

Algorithme 2 Algorithme Grow
fonction Grow(Profondeur, ProfondeurMaximal)
si Profondeur= ProfondeurMaximal **alors**
    retourne un terminal au hasard
sinon
    f <- fonction ou terminal au hasard
    si f est une fonction **alors**
      pour chaque argument a de f faire
        a <- Grow(Profondeur+1, ProfondeurMaximal)
      fin pour

```
        fin si
        retourne f
fin si
fin fonction
```

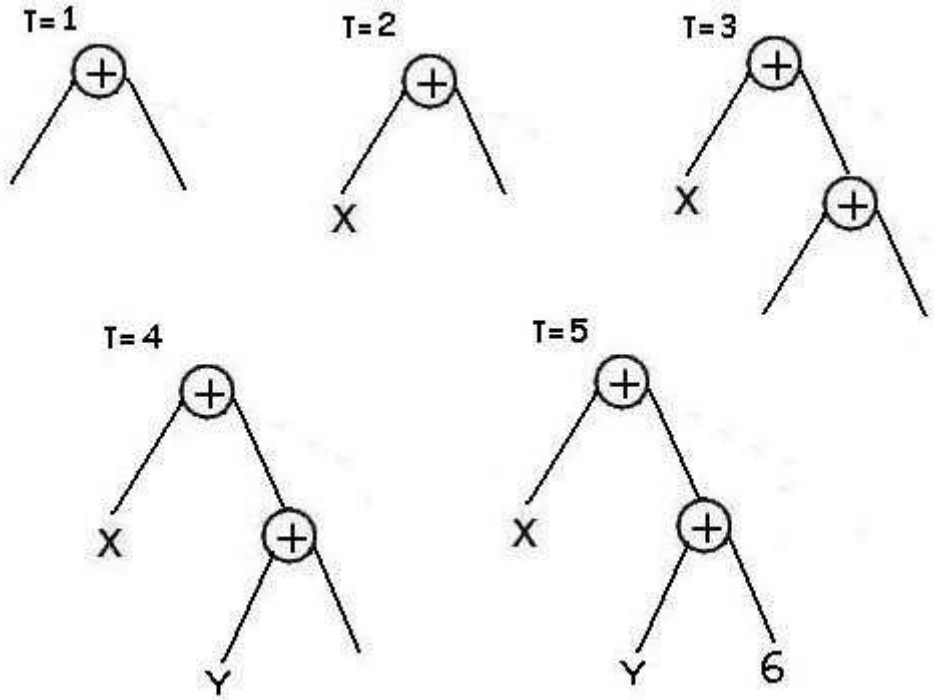

Figure 4: création d'un arbre de profondeur 2, avec l'algorithme grow, le choix du terminal en T=2 a provoqué l'arrêt du développement de la branche gauche même si la profondeur maximale n'est toujours pas atteinte (T=Temps).

La méthode « *full* » crée par contre des arbres ayant systématiquement la profondeur maximale. Dans ce cas, une initialisation avec la méthode « full »implique que tous les arbres de la première génération auront même forme et même nombre de nœuds. La figure 5 montre l'exemple de création d'un arbre de profondeur 2 avec cette méthode. Tant que la profondeur limite n'est pas atteinte, le choix des nœuds est restreint à l'ensemble des fonctions et quand la limite est atteinte, on ne choisit que des terminaux.

C'est pour combler ce manque de variété des formes d'arbre des deux méthodes que Koza [Koza1992] a proposé une combinaison des deux méthodes qui favorise la diversité : « Ramped half-and-half », qui utilise la méthode « full » pour la construction de la moitié de la population et la méthode « grow » pour la deuxième moitié.

L'usage de ces méthodes est très répandu, mais l'estimation de la distribution statistique des principales propriétés, taille et forme des arbres, reste difficile à cerner ; la taille ou la forme d'un arbre dépend largement de l'ensemble des fonctions et des terminaux. Si le nombre de terminaux est largement plus important que l'ensemble des fonctions les arbres « grow » risquent d'être courts et si l'inverse est vrai la taille, la forme des arbres sera

similaire à celle produite par l'algorithme full.

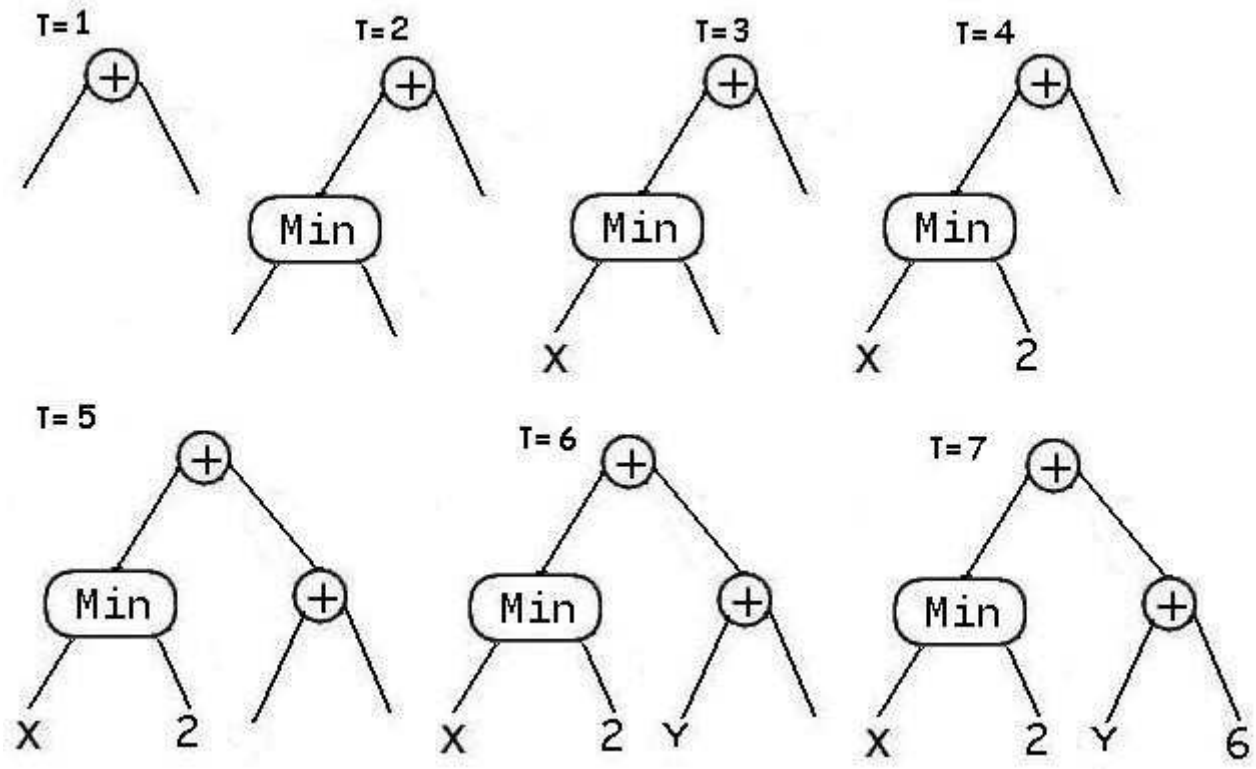

Figure 5: Création d'un arbre plein avec la méthode full.

1.3.3 Les opérateurs génétiques

Les opérateurs génétiques permettent de créer de nouveaux individus au cours de l'évolution. On retrouve généralement deux types d'opérateurs : les opérateurs de croisement et les opérateurs de mutation. Les opérateurs de croisement combinent deux individus parents pour créer un ou plusieurs nouveaux individus enfants avec des caractéristiques « proches » de celles des parents. Quant aux opérateurs de mutation, ils modifient un individu parent pour créer un enfant, modifications supposées plus légères que le croisement.

Ces deux opérateurs provoquent une dualité appelée exploration/exploitation, car il faut en toute étape de l'algorithme effectuer un compromis entre *explorer* l'espace de recherche pour éviter de stagner dans un optimum local et *exploiter* les meilleurs individus obtenus afin d'atteindre de meilleures valeurs en intensifiant la recherche. Ainsi l'opérateur de croisement et l'opérateur de sélection sont des outils d'exploration, tandis que l'initialisation et la mutation sont des outils d'exploitation.

1.3.3.1 Croisement

L'opérateur de croisement est généralement un opérateur binaire. Il associe deux individus appelés parents pour en donner deux nouveaux individus appelés enfants. Cet

opérateur est adapté à la programmation génétique, car il utilise une recombinaison de sous-arbres pour combiner le patrimoine génétique des programmes. Un point de croisement (un nœud), est choisi aléatoirement dans les deux programmes de manière indépendante.

Le programme enfant sera créé à partir d'une copie d'un des parents, mais on remplace au point de croisement le sous-arbre par un une copie du sous-arbre du deuxième parent, sous-arbre dépendant du point de croisement du deuxième parent. La figure 6 montre un exemple de cette opération. La copie des arbres permet de garder intact le programme original, ce qui permet la réutilisation du même parent pour la création de plusieurs nouveaux enfants. Notons également qu'il existe de nombreuses versions du croisement, l'une d'elles par exemple combine deux arbres pour ne conserver qu'un seul enfant.

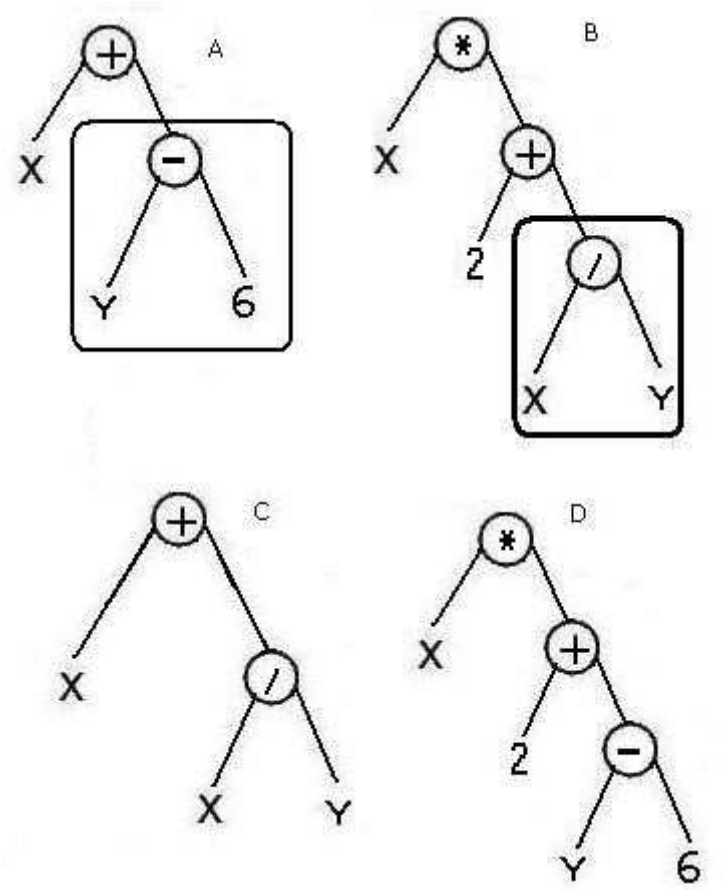

Figure 6: Croisement des deux sous arbres entre les arbres (A) et (B) pour donner (C) et (D) arbres enfants.

Le choix du point de croisement est souvent non uniforme, la sélection uniforme de points pour le croisement ne permet pas un grand échange de patrimoine génétique. Parfois, le croisement peut se réduire à une opération d'échange de feuilles. Pour éviter dans les arbres d' importante taille que le croisement ne constitue qu'un échange de feuilles, Koza a proposé [Koz 1992] de tirer 90% de points de croisement parmi l'ensemble de fonctions et 10% parmi les feuilles.

Un problème très connu en PG est l'accroissement incontrôlé de la taille des programmes, souvent dû au croisement, qui conduit à un effet qu'on nomme effet de congestion (bloat), qui ralentit le processus de programmation génétique. De nombreuses solutions ont été proposées afin de limiter les conséquences de cet effet, parmi elles, on peut citer la limitation du croisement sur des arbres de forme semblable[lan 2000].

1.3.3.2 Mutation

Cet opérateur permet une exploitation plus fine de l'espace de recherche que l'opérateur de croisement. La forme la plus utilisée de cet opérateur consiste à tirer un point au hasard puis remplacer ce nœud par un nouveau sous-arbre, figure 7. Cette mutation est appliquée parfois comme un croisement entre un arbre est un nouvel arbre généré arbitrairement. On nomme cette opération le headless-chicken [Ang 1997].

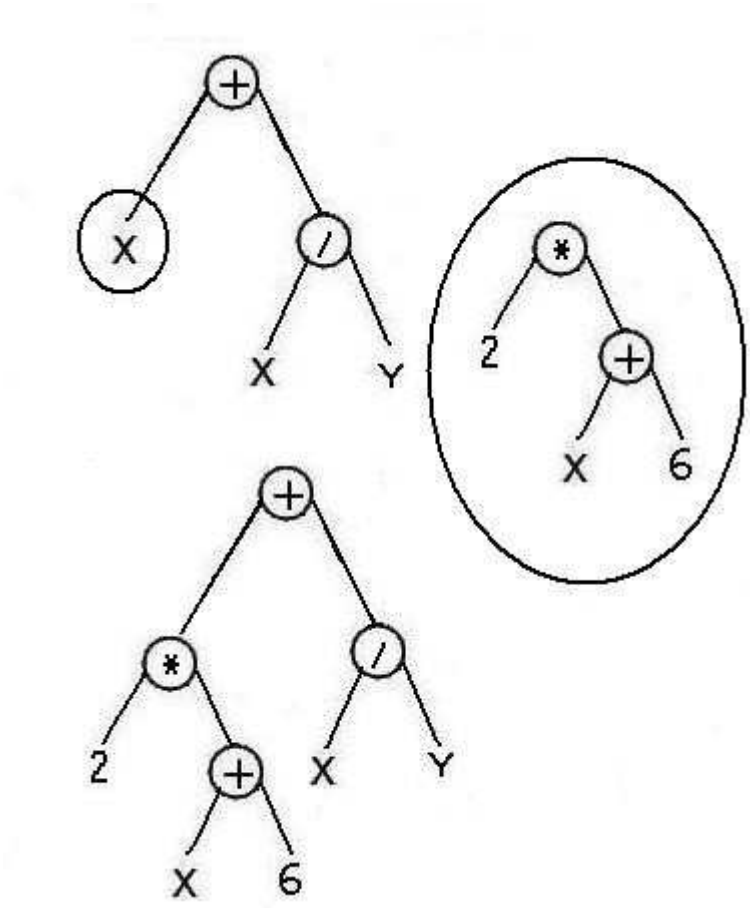

Figure 7: le nœud X est remplacé par le sous-arbres généré aléatoirement dans le cercle

Une autre méthode très utilisée est la méthode dite « point-mutation », semblable à la mutation bit à bit ou bit-flip utilisée pour les algorithmes génétiques [Gol 1989]. On choisit un point au hasard et on le remplace par une primitive du même arité, s'il n'y a pas d'autres primitives de même arité, on ne change rien et on peut passer à d'autres nœuds.

La mutation de sous arbre implique la modification d'un seul sous arbre du programme, mais la mutation par point quant à elle conserve la structure de l'arbre, mais remplace souvent plusieurs nœuds d'un seul programme. Cette mutation est appliquée selon une probabilité par nœud, indépendamment de la taille de l'arbre.

Typiquement, la probabilité de l'application de l'opérateur de croisement se situe autour de 90%, quant à la probabilité de choix de l'opérateur de mutation est autour du 1%. Il est possible d'utiliser un autre opérateur nommé recopie (reproduction operator) avec une probabilité 1-p, où p est la somme des probabilités d'utilisation des opérateurs mutation et croisement.

1.3.3.3 Recopie

La recopie est un opérateur qui n'est autre qu'une simple recopie d'un programme d'une génération à la suivante. Il peut devenir un outil d'élitisme qui a pour but d'éviter la perte éventuelle du ou des meilleurs individus au cours de l'application des opérateurs génétiques (croisement et mutation). Cette procédure très classique a pour avantage de garantir, à coup sûr, la survie des individus les mieux adaptés. Cet outil représente cependant une « arme à double tranchant », parce que l'on favorise l'exploration locale au détriment de la recherche globale, pouvant induire une convergence vers un optimum local.

1.3.3.4 Sélection

Si pour créer une nouvelle génération de programmes, on doit appliquer des opérateurs génétiques tels que le croisement et la mutation, il faut avant choisir les individus sur lesquels on applique ces modifications ; cette opération se nomme la sélection. Elle est stochastique et souvent basée sur la fonction objectif ou fitness, mais l'utilisation d'autres mécanismes reste possible.

La sélection par tournoi est la méthode principalement utilisée, elle consiste à choisir un nombre de programmes aléatoirement de la population, le gagnant de ce tournoi va

être choisi. Pour un croisement qui nécessite deux parents, deux tournois sont organisés. La sélection d'un individu dans un tournoi se fait en choisissant le meilleur entre deux ou plusieurs programmes en les comparant un par un (le nombre d'individus participant au tournoi est nommé taille du tournoi ou pression de sélection). Une pression de sélection régulière évite une invasion de la génération suivante d'un seul très bon programme et ces fils, ce qui peut avoir pour effet la perte de la diversité génétique et peut induire à de mauvais résultats.

Cette sélection aléatoire parmi les candidats au tournoi apporte une certaine diversité, car même s'il préfère le meilleur des individus, le tournoi permet également aux individus de qualité moyenne d'engendrer des enfants. C'est la méthode la plus utilisée en programmation génétique.

Une autre méthode de sélection est le « rank selection » (ou sélection par le rang)[Bak 1985] qui classe les individus selon leur valeur de fitness. L'individu le mieux classé va être choisi avec un nombre de fois plus important que l'individu le plus mauvais.

Il existe également d'autres méthodes de sélection dans la littérature : la roulette (« Roulette wheel ») métaphore de la roulette de casino, où le nombre de cases de la roulette est égal au nombre des individus et leur largeur est proportionnelle à la fitness de ces individus. L'individu sur lequel la boule s'arrête est sélectionné. Une autre variante de cet opérateur est connue sous le nom de *stochastic universal sampling* [bak 1987]qui garantit une sélection avec un biais minimum, où le biais définit la différence entre le nombre de fois où un individu devrait être sélectionné et le nombre de fois où il a été sélectionné. [gol 1989] présente d'autres opérateurs.

1.3.4 Diversité de la population.

La pression de sélection exercée durant l'étape de sélection risque de provoquer une certaine ressemblance entre les individus de la population, ce qui se traduit par une réduction de la diversité de la population et par conséquent un risque de convergence prématurée du système vers un optimum local. Pour éviter cette convergence prématurée, il existe plusieurs techniques pour maintenir la diversité génétique, le surpeuplement ou « crowding » [Dej 1975] qui tend à remplacer des individus « trop »similaires, l'approche de sous-populations qui s'inspire des niches écologique ; cette technique permet de faire évoluer des populations de manière indépendante au sein d'une même génération, celui des îlots[Gor 1992] où chaque sous-population est isolée des autres sous-populations.

1.4 Application à la programmation génétique

Pour appliquer le moteur de programmation génétique il faut prendre certaines décisions pour à certains paramètres :

- Le choix de l'ensemble de fonctions ;

- Le choix de l'ensemble de terminaux ;
- Comment définir la fonction fitness ;
- Quel(s) critère(s) d'arrêt choisir.

1.4.1 L'ensemble de fonctions

La programmation génétique ne fait évoluer que des programmes, mais rarement avec des langages de type *Turing-complet* comme (C, C++, Java, …). En effet, les programmes évolués à l'aide de la programmation génétique sont généralement des expressions et formules exprimées dans des langages spécifiques au domaine du problème posé. La première étape pour spécifier le langage consiste à définir l'ensemble des fonctions et l'ensemble des terminaux, ces ensembles vont donner les ingrédients de base pour construire les programmes.

L'ensemble des fonctions choisies dépend de la nature du domaine du problème. Par exemple pour un problème numérique simple, on peut se contenter d'un ensemble de fonctions arithmétiques (+, -, *, /). mais toute fonction utilisée dans les langages de programmation peut être utilisée. [Wan 2003] a proposé deux caractéristiques des meilleurs ensembles des fonctions, primo des ensembles de fonctions qui contiennent des fonctions similaires aux fonctions recherchées dans les problèmes de régression symbolique et secundo des ensemble qui contiennent le moins possible de fonctions. Le tableau 1.a montre un certain nombre d'exemples des fonctions utilisées

Primitives	exemple
Arithmétiques	+, *, -
Mathématiques	sin, cos, exp
Booléennes	AND, OR, NOT
Conditionnel	IF-THEN-ELSE
boucle	FOR, Do-Until

Tableau1.a Types de fonction

Primitives	exemple
Variables	X, Y, Z
Constates	4, 0.25
Fonctions avec arité 0	rand

Tableau 1.b Types de terminal

1.4.2 L'ensemble de terminaux

Cet ensemble peut contenir comme dans le tableau 1.b, des variables, canstantes ou fonction :

Des variables : Des entrées du programme.

Des fonctions : ces fonctions d'arité nulle retournent une valeur au hasard chaque utilisation (comme la fonction rand), des informations comme la fonction d'un capteur de robot qui va retourner la distance qui le sépare d'un obstacle... De telles fonctions peuvent également avoir un effet de bord (side effect functions), car en plus de retourner une valeur, elles peuvent engendrer des changements dans leur environnement, comme afficher ou dessiner sur l'écran, contrôler un robot...

Des constantes : prédéfinie en général, elles sont générées aléatoirement dans le processus d'initialisation. On parle dans ce cas d'ERC (pour Ephemeral Random Constants)

Notons que l'utilisation de rand rendra un résultat différent d'une exécution à l'autre du programme, même si les variables ont les mêmes valeurs d'entrée à chaque fois.

Selon Koza, les ensembles de fonctions et de terminaux doivent être choisis de manière à satisfaire deux conditions : la clôture et la suffisance.

1.4.3 La clôture

La clôture est une condition définie par Koza [1992] par la satisfaction de deux conditions : l'unicité de type et la satisfaction de l'évaluation.

Le choix d'utiliser un seul type de données que retournent les noeuds (fonctions et terminaux) est nécessaire suite aux opérations génétiques aléatoires qu'effectuent la mutation ou le croisement ; tout terminal peut devenir argument potentiel d'une fonction de notre ensemble. Par exemple, pour des fonctions arithmétiques comme -, +, /, et *, l'utilisation du type entier pour les arguments et pour la valeur du retour est nécessaire afin d'autoriser la composition de ces fonctions. Il est également possible de faire des conversions automatiques de type. Par exemple, pour le type booléen, on peut considérer qu'une valeur positive est vraie et une valeur négative est fausse. Il faut cependant éviter un éventuel biais.

Il existe un moyen de contourner cette condition d'unicité de type, qui est de ne permettre une opération génétique que si elle donne un nouvel arbre ou sous-arbre qui retourne une valeur qui possède le même type que le sous-arbre remplacé.

L'autre condition de clôture est l'infaillibilité de l'évaluation « safety evaluation », car des programmes peuvent échouer dans la phase d'évaluation : par exemple, en rencontrant une division par 0, ou une fonction MOVE_FORWARD d'un robot qui se trouve déjà face au mur. Une solution qui règle le problème est la modification de ces fonctions de façon à ne pas produire ce genre de comportement indésirable. Des versions protégées des fonctions numériques sont souvent utilisées, elles consistent à tester si les arguments reçus peuvent engendrer un problème. Dans ce cas, on donne une valeur prédéfinie de la fonction, sinon la fonction suit son processus normal. Ainsi

la fonction de division protégée, notée % : si le dénominateur est de valeur nulle, par convention la valeur retournée est 1 ; de la même façon, la fonction MOVE_AHEAD ne produira aucune modification si ce déplacement est illégal pour le robot. Une autre solution alternative consiste à mettre des exceptions et réduire le fitness des programmes qui déclenchent de telles exceptions.

1.4.4 la suffisance de l'ensemble des primitives

Une autre propriété nécessaire est que la solution construite par les ensembles de terminaux et de fonctions permettent d'exprimer une solution qui permettra de résoudre le problème. Malheureusement, ceci n'est garanti que dans le cas de problèmes résolus en théorie ou résolus avec d'autres méthodes.

Par exemple, l'ensemble {NOT, AND, OR, x1, x2, ...} est suffisant pour construire une solution de n'importe quel problème booléen de variables x1,...xn. Autre exemple, un ensemble tel {+, -, *, /, x, 0, 1, 2} n'est pas suffisant pour représenter des fonctions transcendantes, car n'importe quelle combinaison de cet ensemble ne peut donner une fonction comme exp(x).

Quand un ensemble de primitives est insuffisant pour exprimer une solution, la programmation génétique se limite à donner des programmes d'approximation de la solution. Notons toutefois que cette approximation peut parfois être une solution acceptable selon les besoins de l'utilisateur, rendant ainsi inutile d'opter pour l'ajout de plus de primitives ; choix qui peut s'avérer coûteux en temps de calcul et introduire un biais inutile et imprévu. Certaines études considèrent le surcoût acceptable [Pol 2008].

1.4.5 La fonction fitness

Si le choix de l'ensemble des primitives détermine l'espace de recherche du problème, c'est-à-dire tous les programmes possibles à partir de composition des éléments de l'ensemble des fonctions et des terminaux, on doit déterminer les régions de l'espace de recherche les plus propices à la résolution au problème posé. C'est pour cela qu'il faut une mesure qui évalue la qualité de chaque programme. En général, chaque programme possède une valeur numérique (fonction fitness) qui détermine sa qualité par rapport au problème posé.

La fitness peut se mesurer de plusieurs manières : une mesure d'*erreur* pour les problèmes de régression symbolique où on calcule la différence entre le résultat obtenu par le programme et la valeur attendue, un *temps* d'exécution pour ramener le système dans un état voulu (cela peut faire référence au coût d'argent ou coût d'énergie), ou une mesure de *précision* dans les problèmes de classification, ou encore

la mesure d'une conformité d'une structure au critère du design donné par l'utilisateur, ou bien la mesure du gain qu'un programme peut faire dans un scénario de jeu, ...

Le fitness d'un individu s'effectue en évaluant son exécution.

Le processus d'interprétation d'un arbre passe par l'évaluation des nœuds de cet arbre, dans un ordre qui garantit qu'aucun nœud ne sera exécuté avant de connaître les valeurs de ces enfants ou arguments s'il en a. Ceci est généralement réalisé en traversant l'arbre de manière récursive, il est également possible de commencer par des feuilles vers la racine. Les deux méthodes sont équivalente dans le cas où aucun nœud de l'arbre n'a d'effet indésirable. La figure 8 montre un exemple qui part du nœud racine vers les feuilles.

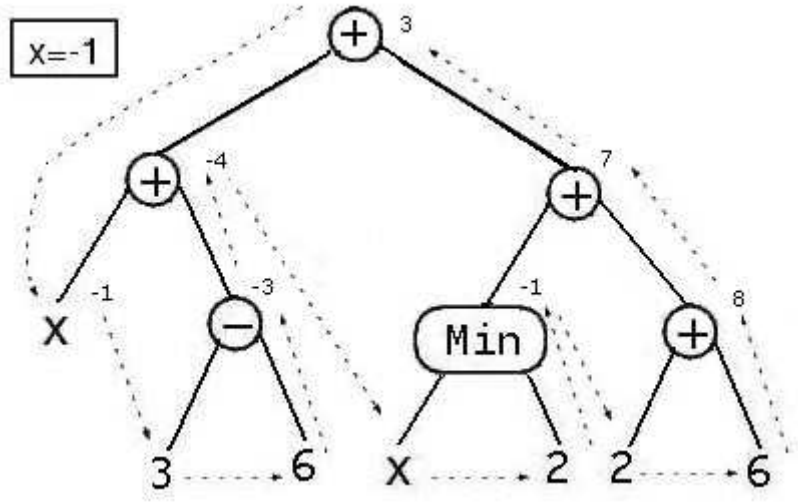

Figure 8: parcours récursif à partir du nœud racine (-) de l'arbre(programme (- (+ x (- 3 6)) (+ (min x 2) (+ 2 6))). la variable x a pour valeur -1. le résultat final de l'exécution du programme est 3.

L'algorithme 3 est une procédure en pseudo-code qui se charge de l'interprétation des programmes.

D'autres méthodes d'interprétation sont aussi possibles pour les machines multiprocesseurs, permettant ainsi de profiter du parallélisme pour accélérer le calcul de la fitness.

Dans certains problèmes, on s'intéresse au résultat final retourné par le programme, suite à l'évaluation de l'arbre. Mais dans d'autres problèmes, le plus important est l'action du programme qui contient des fonctions avec des effets de bord ; le fitness dans ce cas va dépendre de l'exécution du programme avec différente entrées, ou différentes conditions. Un programme de contrôle de robot, par exemple va être testé avec différentes positions de départ. Tous ces résultats sont appelés cas de fitness ou fitness cases et vont contribuer à la valeur de fitness.

La PG est également capable de résoudre des problèmes multi-objectifs qui combinent deux ou plusieurs objectifs souvent contradictoires. Ce domaine de recherche est très actif en PG[Deb 2001, Bar 2004, Cur 2007].

1.5 Les paramètres de la PG

De nombreux paramètres contrôlent la programmations génétique. Un des paramètres primordiaux est la taille de la population. D'autres paramètres contrôlent les probabilités de choix des opérateurs génétiques ou la taille maximale des programmes pour restreindre l'espace de recherche et diminuer l'effet de bloat. Comme tout algorithme évolutionnaire, il faut noter qu'il n'existe pas de réglage optimal pour tous les problèmes de PG. Notons cependant que le système est relativement robuste et différentes configurations des paramètres peuvent fonctionner et donner des résultats acceptables et comparables. En outre, un consensus s'est dessiné ces dernières années montrant qu'en général une population importante de l'ordre de plusieurs milliers, dizaines ou centaines de milliers d'individus avec un nombre de génération faible (de l'ordre de la centaine) offre les meilleures performances.

```
procedure: eval( n )
1: si  n est une liste et alors
2:   proc = n(1) {Non terminal: extraire la racine}
 3:  si proc est une fonction alors
4:     valeur = proc( eval(n(2)), eval(n(3)), ... ) {Fonction: évaluer les arguments}
5:   sinon
6:   valeur = proc( n(2), n(3), ...) {Macro: ne pas évaluer les arguments}
7:   fin si
8:  sinon
9:     si n est variable ou n est constante alors
10:       valeur = n {variable ou constante: lire la valeur}
11:    sinon
12:       valeur = n() {fonction sans argument: exécuter}
13:    fin si
14:  fin si
 15: retourne valeur
```

algorithme 3: interpréteur de la programmation génétique

Notons que n est une notation préfixée, n(1) est la racine de l'expression, n(2) et n(3) sont le premier et le deuxième argument de la racine n(1).

Comme on a indiqué précédemment, il est commun de créer la population initiale à l'aide de l'algorithme *ramped half-and-half* avec une profondeur initiale des arbres variant dans un intervalle entre deux et six.

Comme on a expliqué précédemment, cette profondeur va croître durant l'évolution. En ce qui concerne la taille de la population, l'espace requis pour le stockage ne pose pas de limitation contrairement au temps d'évaluation requis par le calcul de la fitness qui nécessite la limitation de la population. En général, la taille de la population minimum est de 5000 individus, mais souvent on préfère des larges tailles de population pour permettre une large diversité dans la population. Certains travaux suggèrent de plus petite taille de population avec un grand nombre d'essais, à peu d'essai avec de large taille de populations. Le nombre de générations est en général fixé entre 50 et 500[Pol 2008].

1.6 Conclusion

Le but de ce chapitre est de donner une vue générale sur les algorithmes évolutionnaires et un survol des principaux éléments de la programmation génétique. Nous allons nous intéresser par la suite au problème de la sélection d'attribut dans le but de l'utiliser pour réduire l'impact des terminaux non pertinents dans la programmation génétique. Comme on a indiqué dans ce chapitre, la taille de l'ensemble des terminaux est un des paramètres qui auront indirectement un impact sur le coût de calcul de la programmation génétique.

2 Chapitre 2 : SELECTION D'ATTRIBUTS ET CLASSIFICATION

2.1 Introduction

Ce chapitre s'intéresse au problème de sélection d'attributs (« feature selection ») ou identification d'attributs. Ce problème qui recouvre plusieurs domaines est considéré comme un point faible des techniques comme l'apprentissage supervisé ou la fouille de données [Koh 1994][Guy 2003]. Une définition généralement admise est la suivante :

« la sélection d'attributs étudie comment choisir un sous-ensemble ou une liste d'attributs ou variables qui sera utilisé pour construire un modèle qui décrit les données. »

La grande taille de données pose de gros défis aux méthodes d'apprentissage en intelligence artificielle, dans le traitement d'images et également en fouille de données, car il est de plus en plus nécessaire de traiter des données décrites par un très grand nombre d'attributs, comme en bio-informatique où les données donnent des niveaux d'expression de milliers de gènes [Dai 2006].

La sélection des attributs s'est imposée dans ce contexte comme un outil fréquemment utilisé pour réduire la taille des ensembles de données. En effet, en supprimant les éléments non pertinents et redondants, on arrive à réduire la probabilité de sur-apprentissage. Différents travaux [Ng 2004, Don 2006] montrent qu'il est possible de supprimer les attributs non pertinents sans détérioration de la performance. En outre, la sélection d'attributs permet de faciliter la visualisation des données et leur analyse.

Dans la suite du chapitre, nous présenterons quelques notions d'apprentissage artificiel, puis nous définirons la notion de pertinence et de sélection d'attributs.

2.2 Classification et sélection d'attributs

La sélection d'attribut s'est développée notamment dans le domaine de l'apprentissage automatique (*machine-learning*), discipline de l'intelligence artificielle où un modèle se construit en s'appuyant sur un processus d'apprentissage. Il existe deux grandes catégories d'algorithmes d'apprentissage : supervisé et non supervisé.

Une technique de base en apprentissage artificiel est celui de la *classification*. *Un classifieur* est une procédure qui prend en entrée une donnée et la classe comme appartenant à une classe en sortie. Le concept de classification est étroitement lié à la notion de partition d'un ensemble fini. La définition qui suit correspond à la notion de classification dite dure :

Définition 2.1: Étant donné un ensemble fini d'objets Ω, on appelle partition de Ω toute famille de parties non vides de Ω disjointes deux à deux dont l'union forme l'ensemble Ω. Alors, si *C* est une partition de Ω, On a :

$$\mathcal{C} = \left\{ \mathcal{C}_i \in \mathcal{P}(\Omega) \setminus \{\varnothing\} : \bigoplus_{i=1}^{K} \mathcal{C}_i = \Omega \right\}$$

Cette définition impose d'une part que tous les objets doivent appartenir à une classe, et uniquement à une classe.

2.2.1 Classification non supervisé

Dans le cadre de la classification non supervisée, on ne dispose pas de classes prédéfinies des éléments à classer et on essaie de regrouper les données de mêmes caractéristiques dans les mêmes classes. Le regroupement de ces attributs est effectué généralement à l'aide de la notion de distance entre les variables, ou de distance par rapport au centre de masse de chaque classe. Les classes ne sont donc pas connues et seront créées selon la ressemblance entre les attributs. Les principales méthodes de classification sont les algorithmes de classification hiérarchique et les méthodes de regroupement [Dai 2006] [Jai 1999] [Gra 2002].

Des algorithmes comme la K-moyenne ou « K-means » permettent de segmenter les données en K groupes homogènes (ou « clusters »). On prend k point dans l'espace de données aléatoirement comme centroïdes initiaux et pour chaque donnée on cherche le plus proche centroïde et on la place dans ce groupe et si aucun changement n'est apporté aux valeurs des centroïdes, l'algorithme prend fin [The 1999] .

L'autre principale méthode de classification non supervisée est la méthode de regroupement, telle que la classification hiérarchique [Jam 1978][Har 1975][Rou 1985] qui a pour objectif de répartir les éléments d'un ensemble en groupes, c'est-à-dire d'établir une partition de cet ensemble. Différentes contraintes sont imposées, chaque groupe devant être le plus homogène possible et les groupes devant être les plus différents possibles entre eux.

En général, ces méthodes sont hiérarchiques et récursives, de types ascendant ou descendant. L'algorithme d'une classification hiérarchique ascendante se déroule généralement en quatre étapes :

1. On considère tous les objets dans l'ensemble de départ comme appartenant à sa propre classe ;
2. On calcule la distance entre les classes suivant un critère de distance ;
3. On regroupe les deux classe les plus proches pour former une nouvelle classe ;
4. Itération des étapes 2 et 3 jusqu'à l'obtention d'une classe unique.

En résumé, cet algorithme consiste à agréger de manière itérative deux individus « les plus proches » en partant d'un ensemble de n singletons et jusqu'à obtenir le nombre de classes souhaité.

2.2.2 Classification supervisée

La classification supervisée a pour objectif d'identifier les classes auxquelles appartiennent des objets à partir de traits descriptifs. Les classes dans ces approches sont connues et on dispose d'exemples de chaque classe (ensemble d'entraînement) qui vont nous aider à déterminer l'appartenance d'un objet.

Parmi les méthodes existantes, les séparateurs à Vaste Marge (SVM) ou Machines à Support Vectoriel introduites en 1992 [bos 1992] restent l'une des méthodes de classification les plus utilisées. Elle repose sur deux concepts clés : la marge maximale et la notion de fonction noyau. On cherche l'hyperplan qui sépare les attributs positifs des négatifs, en garantissant que la distance entre la frontière de séparation et les attributs les plus proches (marge) soit maximale, c'est ce qu'on appelle les vecteurs supports (Figure 1). Dans le cas où les données ne sont pas linéairement séparables, la deuxième idée intervient pour transformer l'espace de représentation des données d'entrées en un espace de plus grande dimension appelé espace de caractéristiques, dans lequel il est possible de réaliser une séparation linéaire. Il existe d'autres méthodes comme les réseaux de neurones, ou les arbres de décision [bre 1984] [Qui 1993].

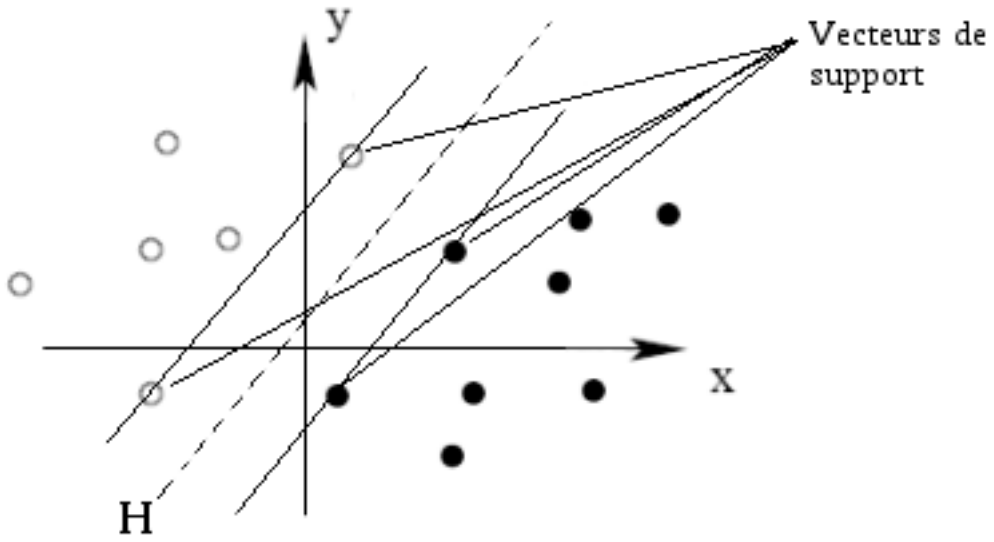

Figure 9: Le Hyperplan H sépare les deux ensembles de points, les deux points les plus proches de chaque groupe sont appelés vecteurs de support.

En général, les systèmes d'apprentissage peuvent être basés sur des hypothèses probabilistes (classifieur bayesien naïf [Fri 1997]) ou sur des notions de proximité (plus proches voisins) ou sur des recherches dans des espaces d'hypothèses (arbres de décision [Bre 1984], réseaux de neurones).

La sélection d'attribut est donc là pour assurer de bonnes performances de classification, car réduire l'ensemble des attributs revient à réaliser un meilleur apprentissage et réduit la complexité. Ceci est également vrai pour les domaine de traitement d'images [fuknga 1990] ou de fouille de donnée (datamining) [singh 2002].

Par la suite, nous allons définir la notion de pertinence et présenter quelques méthodes de sélection d'attributs de la littérature.

2.2.3 la pertinence

On trouve différentes définitions de la pertinence dans la littératures, car celles-ci peuvent être adaptées selon le domaine sur lequel elles sont appliquées. Voici quelques unes des définitions les plus intéressantes pour notre étude :

On définit le rapport à un concept cible.

Définition 2.2 : L' attribut F_i est pertinent pour un concept cible *C*, s'il existe une paire d'*exemples* A et B tels qu'ils ne sont différents qu'en la valeur de l'attribut F_i et $C(A) \neq C(B)$.

Par exemple si on a un problème de classification avec 4 d'attributs descripteurs F1, F2, F3, F4, on suppose que la fonction de classification s'écrit C=g(F1, F3). On remarque que pour déterminer la fonction cible seuls F1 et F3 sont indispensable, F2 et F3 peuvent être éliminés car ils ne sont pas utiles pour définir C. {F2, F3} est le sous ensemble d'attributs optimal pour ce problème.

Kohavi [Koh 1994] [Koh 1997] a proposé un renforcement de la pertinence en introduisant deux notions, fortement pertinent et faiblement pertinent.

Selon [koh 1997], les attributs pertinents sont ceux dont les valeurs varient systématiquement avec les valeurs de classe. Soit par exemple F un ensemble d'attributs, F_i un attribut, et S_i = F - { F_i }. On suppose que l'on travaille avec un espace probabilisé *P,* où *P*(C|S) est la probabilité de la classe C sachant les attributs de l'ensemble *S*.

Définition 2.3 : un attribut Fi est fortement pertinent *si et seulement si :*

$$P(C/F_i, S_i) \neq P(C/S_i)$$

Définition 2.4 : un attribut Fi est faiblement pertinent *si et seulement si* :

$$P(C/F_i, S_i) = P(C/S_i) et \exists S'_i \subset Si \, tel \, que \, P(C/F_i, S'_i) \neq P(C/S'_i)$$

Et finalement la notion de non pertinence

Définition 2.4 : un attribut Fi est non pertinent *si et seulement si* :

$$P(C/F_i, S_i) = P(C/S_i) et \forall S'_i \subseteq Si \, tel \, que \, P(C/S'_i) = P(C/S'_i)$$

Ainsi, un sous ensemble d'attributs optimal doit contenir des attributs fortement pertinents, leur absence peut causer un défaut de reconnaissance de la fonction cible. Un attribut faiblement pertinent n'est pas toujours important.

La non-pertinence se définit simplement à partir des définitions de forte et faible pertinence ; ces attributs ne sont pas nécessaires dans un sous ensemble optimal d'attributs.

Notons qu'il se peut qu'il existe des attributs qui fournissent les mêmes informations, ceux-là sont qualifiés de redondants.

2.2.4 La sélection des attributs

La sélection d'attributs consiste à choisir entre les éléments d'un ensemble d'attributs de grande taille des sous-ensembles d'attributs ou des listes d'attributs jugés intéressants pour le problème traiter. Dans l'idéal, on cherche à réduire le coût de traitement des grandes dimensions des données en éliminant des attributs non pertinents, ou redondants et on chercher à identifier un nombre d'attributs de taille minimale nécessaire et suffisant pour définir le concept cible.

Liu [Liu 2005] définit la sélection d'attributs comme un processus de sélection des attributs originaux, où on peut mesurer par des critères d'évaluation si un sous ensemble est optimal. Il a introduit une taxonomie dans le processus de recherche de la sélection d'attributs : les stratégies de recherche où les algorithmes se divisent en sous-catégories recherche complète, séquentielle ou aléatoire, les critères d'évaluation sur les méthodes (filtre, enveloppe) et en troisième point les techniques de fouille de données car, la disponibilité des informations sur les classes dans les tâches de classification ou « *Clustering* » affecte les critères d'évaluation utilisées par les algorithmes de sélection d'attributs.

En général, dans la littérature, on préfère séparer les méthodes de sélection de caractéristiques en deux approches filtre et enveloppe (Wrapper), selon leur dépendance ou indépendance à l'algorithme d'apprentissage [Liu 2005]. Les méthodes de l'approche filtre exploitent les propriétés propres des attributs utilisées sans référence à une quelconque application. Les méthodes de type enveloppe définissent la pertinence des caractéristiques par l'intermédiaire d'une prédiction de la performance du système final. Les méthodes de l'approche filtre sont généralement moins coûteuses mais aussi moins efficaces que les secondes.

La sélection d'attribut peut être vue également suivant le type d'évaluation des attributs. Les procédures les plus utilisées consistent à identifier un sous-ensemble de nombre minimum d'attributs qui satisfont un critère d'évaluation, et la deuxième catégorie qui cherche à évaluer les attributs ou variables de manière individuelle, ce qui peut permettre de classer les attributs selon un critère de pertinence (« *ranking feature » [Guy 2003]*) et de choisir les attributs les mieux classés. Parmi les méthodes de sélection d'attributs qui utilisent ce principe on a la mesure des poids des attributs (« *weighting method »).*

2.2.4.1 La sélection d'attribut comme un problème d'optimisation

Le problème de la sélection d'attributs peut être vu comme une recherche dans un espace d'hypothèses (ensemble de solutions possibles) [blu 1997]. Étant donné un ensemble initial S de n attributs, la sélection d'un « bon » sous-ensemble d'attributs nécessite d'examiner potentiellement 2^n-1 sous-ensembles possibles. La qualité d'un sous-ensemble sélectionné est évaluée selon un critère de performance que l'on notera $\Im$. Dans le cas d'un problème de classification supervisée, ce critère est très souvent la précision d'un classifieur construit à partir de l'ensemble des attributs sélectionnés.

La recherche d'un sous-ensemble d'attributs, optimal pour le critère $\Im$ que l'on s'est donné, est un problème NP-difficile [Dav 1994]. Plusieurs approches peuvent être envisagées pour contourner cette difficulté. Elles sont formalisées dans la définition suivante :

Définition 2.5 : Sélection d'attributs) [Mol 2002] Soit X un ensemble d'attribut. Soit $\Im$ une mesure d'évaluation qui attribue à tout sous-ensemble de X un score : $\Im$: $\bar{X} \subseteq X \to \Re$. $\Im$ doit être optimisée (maximalisée ou minimisée suivant la nature de $\Im$), on suppose dans la suite que $\Im$ doit être maximisée. Plusieurs techniques sont possibles :

- Nombre d'attributs fixé : pour un nombre m fixé, avec m<n, on cherche à trouver $\bar{X} \subseteq X \to \Re$ tel que |X| = m et que $\Im(\bar{X})$ soit maximum ;

- Seuil de performance fixé : on se donne une valeur seuil $\Im_{opt}$, c'est-à-dire, le minimum acceptable pour $\Im$, est on cherche à trouver $\bar{X} \subseteq X$ tel que le cardinal de X soit le plus petit possible et que $\Im(\bar{X}) \geqslant \Im_{opt}$;
- Compromis performance et nombre d'attributs : il faut trouver un compromis entre le fait de minimiser le nombre d'attributs $|\bar{X}|$ et le fait d'optimiser $\Im(\bar{X})$.

2.2.4.2 Approches de la sélection d'attributs

- **Procédure générale de la sélection d'attributs.**

Généralement, les techniques de sélection d'attributs analysent les sous-ensembles des attributs, et essaient de trouver le meilleur sous-ensemble d'attributs parmi les 2^n sous-ensembles candidats possibles. Cette procédure s'avère évidemment très coûteuse, même pour des ensembles de taille modeste d'attributs. Des méthodes tentent de réduire ce coût en installant des critères d'arrêts de recherche ; ceci peut passer par un compromis de performance et selon [Dash 1997][Blu 1997], une méthode de sélection suit en général quatre étapes basiques (figure 2.1):

1. Une procédure de génération de sous-ensemble candidat potentiel ; c'est-à-dire déterminer le point de départ dans l'espace de recherche (choix entre approches « *forward selection* » stratégie ascendante ou « *backward selection* » ;
2. Le choix de stratégie d'évaluation des sous-ensembles d'attributs ;
3. Critère d'arrêt de la recherche ;
4. Procédure de validation des sous ensembles.

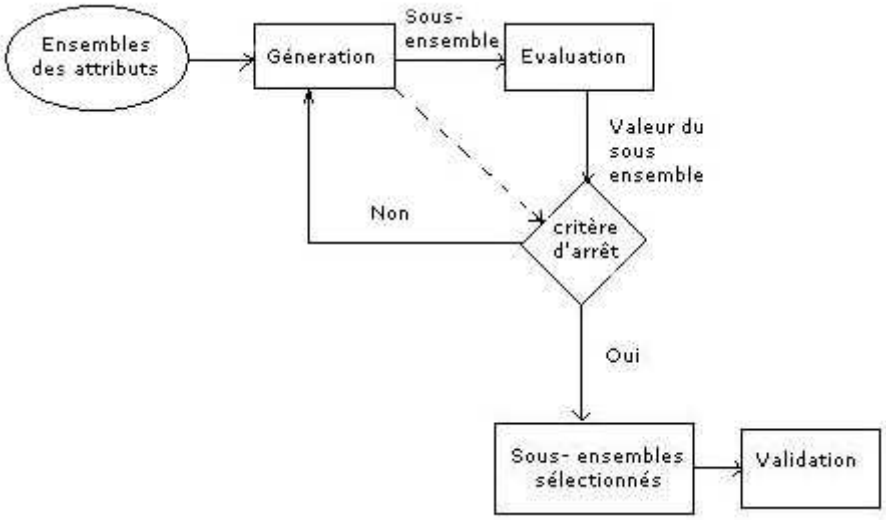

Figure 10: étapes basiques de la méthode de sélection d'attributs

La procédure de génération est une procédure de recherche[Lan 1994][Liu, 1998.] qui donne des sous-ensembles « départ » d'attributs afin qu'ils soient évalués suivant un critère d'évaluation. La procédure de recherche de sous-ensembles commence par le choix suivant : soit partir d'un ensemble qui contient tous les attributs qui seront éliminés de manière itérative par la suite (« *backward aproach* ») soit commencer par un ensemble vide et ajouter des attributs au fur et à mesure. On peut éventuellement choisir une méthode qui utilise les deux techniques qui ajoute et supprime les attributs simultanément. Dans l'étape de l'évaluation, la fonction mesure la qualité du sous-ensemble et le compare avec le précèdent meilleur sous-ensemble. S'il est meilleur, il le remplace. Le critère d'arrêt va stopper ce dernier processus de recherche dans l'espace des sous-ensembles possibles. Ce critère peut être un nombre d'attributs ou un nombre d'itérations atteint. L'étape de validation teste la validité du sous ensemble choisi en comparaison avec les résultats de méthodes différentes.

Les trois principales techniques de sélection d'attributs sont présentées :

– **Les approches filtre**

Dans les approches filtre ou « *filter methods* », la sélection des attributs s'effectue avant le déroulement de l'algorithme d'induction.

Le principe consiste à évaluer chaque attribut et lui donner un score de pertinence, score qui permet le classement des attributs avant la sélection des attribut les plus pertinents. Il est également possible de donner un score à des sous-ensembles d'attributs.

Les méthodes de filtre sont de complexité raisonnable, et peuvent être utilisées avec un nombre d'attributs important. Cette méthode ne peut être utilisée en cas de besoin d'élimination des attributs redondants car cette méthode n'accorde pas d'importance aux interactions possibles avec les autres attributs.

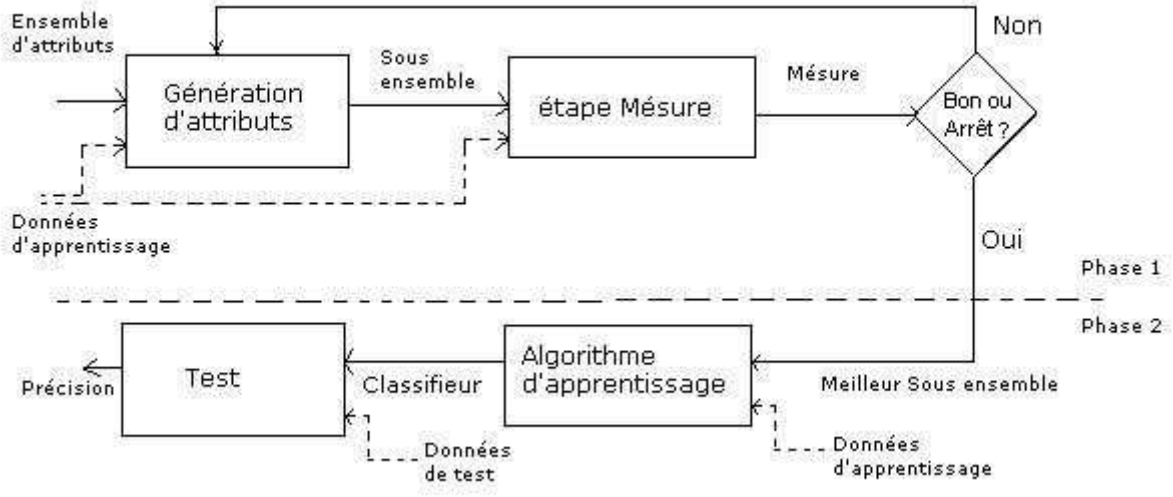

Figure 11: méthode filtre

La figure 11 montre le déroulement général des techniques basées sur les filtres.

– **Les approches enveloppes**

Contrairement aux approches filtre qui ignorent l'influence des variables sélectionnées sur la performance de l'algorithme d'apprentissage, les approches « enveloppantes » ou enveloppes (« *wrappers methods* ») utilisent l'algorithme d'apprentissage comme une fonction d'évaluation.

Le mécanisme de sélection des méthodes enveloppe utilise la performance en aval pour la sélection des attributs. Dans les problèmes d'apprentissage, on interagit avec un classifieur pour trouver un sous-ensemble d'attributs optimal [koh 1997]. En conclusion, la sélection peut être vue comme une exploration de sous-ensembles candidats et le processus d'apprentissage effectuera la tâche d'évaluation des sous-ensembles d'attributs.

Le désavantage majeur de cette approche vis a vis des méthodes filtres est le temps de calcul nécessaire pour construire un classifieur nécessaire pour l'évaluation de chaque sous-ensemble candidat et il existe également un risque de sur-apprentissage.

La figure 12 montre le déroulement général des techniques d'enveloppe.

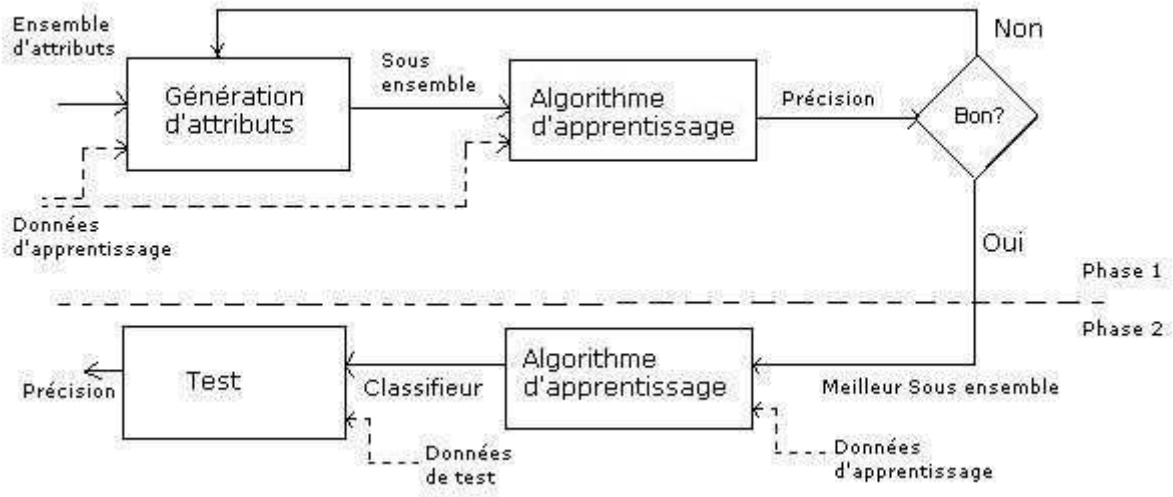

Figure 12: méthodes enveloppe

– **Les approches intégrées**

Les approches intégrées (« embeded methods ») exécutent la sélection d'attributs durant le processus de l'apprentissage. Le processus de la sélection de variables est effectuée parallèlement au processus d'apprentissage. Le sous-ensemble de variables ainsi sélectionné sera choisi de façon à optimiser le critère d'apprentissage utilisé. Ces méthodes sont proches des méthodes enveloppes (car elles combinent le processus d'exploration avec un algorithme d'apprentissage). Le classifieur sert en plus d'évaluation du sous-ensemble candidat, à guider le mécanisme de sélection. Des méthodes comme l'algorithme C4.5 [Qui 1993], ID3[Qui 1983] ou CART [Bre 1984] ont la

même approche car la sélection d'attributs se fait en même temps que la construction du modèle.

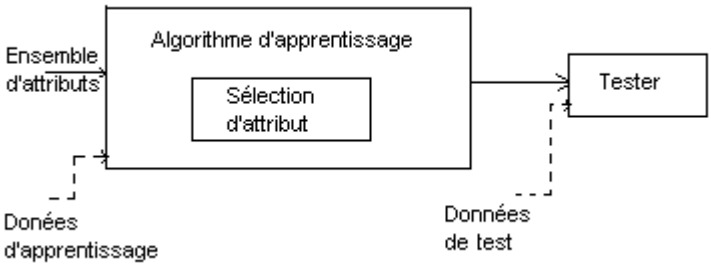

Figure 13: l'approche intégrées, où le classifieur évalue les sous ensembles candidat et guide le mécanisme de sélection.

L'avantage de ces méthodes et que le classifieur guide le processus de recherche par les informations qu'il fournit, ce qui peut les rendre plus efficace que les méthodes d'enveloppes [Say 2007].

– **Méthodes de mesure du poids des attributs.**

L'approche qui nous intéresse le plus est celle de l'évaluation individuelle, l'approche de mesure du poids de chaque attribut. Cette approche qualifiée d'intègre, car au lieu de chercher à optimiser un sous-ensemble, on cherche à appliquer des fonctions qui donnent un degré de pertinence à tous les attributs durant l'apprentissage, une valeur de poids élevé pour les terminaux pertinents et un degré bas et proche de zéro pour les attributs non pertinents. Cette technique est plus facile à implémenter en apprentissage incrémental [Gir 2000], et est motivée par des considérations de performance [Blu 1997].

Un des algorithmes les mieux connus par l'attribution de poids aux attributs est l'algorithme Perceptron [Min 1969] dans le domaine des réseaux de neurone. Il possède une règle de mise à jour de poids qui ajoute ou soustrait en réponse à l'erreur perçue en apprentissage. L'algorithme Winnow [Lit 1988] a amélioré la règle de mise à jour du perceptron, en optant pour une mise à jour de manière multiplicative, au lieu d'additive.

2.3 Conclusion

Nous avons présenté dans ce chapitre les deux approches « classiques » de classification, la classification supervisée et la classification non supervisée. Nous avons vu la notion de pertinence ainsi que les approches heuristiques de sélection d'attributs, ainsi que l'interaction avec la classification et également nous avons introduit l'approche de mesure de poids des attributs qui sera utilisée au chapitre suivant.

3 Chapitre 3: Mutation et Sélection de terminaux pertinents

Dans ce chapitre nous allons étudier les différents moyens pour améliorer la régression symbolique avec la programmation génétique en utilisant une méthode de sélection de terminaux pertinents.

3.1 Régression symbolique

La régression symbolique ou identification de fonctions, est une méthode de modélisation mathématique et d'analyse des données numériques, qui nécessite de trouver une approximation d'une fonction mathématique sous une forme symbolique. On essaie à partir d'un ensemble de valeurs expérimentales, de trouver la courbe qui reproduit le mieux les variations de la grandeur à étudier et qui passe le plus près possible des points dont on dispose, de manière à déterminer le comportement général du système.

En d'autres termes, la régression symbolique est une méthode de recherche dans l'espace des expressions mathématique qui minimise les métriques d'erreurs.

La régression symbolique essaie donc de trouver des fonctions dont la sortie possède certaines propriétés désirées. On essaie dans la plupart des cas, de trouver une fonction qui coïncide avec les points donnés sans s'intéresser à la structure de la fonction.

Il existe plusieurs méthodes classiques pour approximer une fonction comme la méthode de régression par polynômes de Bernstein [Sta 1986] [Bro 1999], la technique de régression d'un ensemble de données ou d'une fonction par un polynôme, introduite par Weinstaier, qui, selon le théorème Stone-Weierstrass stipule que « toute fonction continue à valeurs réelles sur un compact de R peut être approximée uniformément par une suite de polynômes. »

Un autre exemple de méthodes de régression est la technique à base de splines [Fri 1991] [Boo 1978] [Sch 1973] qui consiste à définir une fonction par morceaux par des polynômes. Cette méthode est souvent préférée à une approche par régression polynomiale (Bernstein ...), car on obtient des résultats similaires en utilisant des polynômes ayant des degrés inférieurs. Mais la régression symbolique offre quelques avantages sur ces approches :

- Le libre choix des points (nombre et emplacement) qui vont servir à approximer la fonction ;
- on n'apprche pas la fonctionseulement à l'aide de polynômes: il est possible d'utiliser n'importe quelle classe de fonctions adaptée à notre problème (fonction trigonométriques par exemple).

La régression symbolique s'intéresse simultanément à la recherche des paramètres comme la régression linéaire ou non-linéaire et à la forme et structure de la fonction.

Cette technique donne automatiquement des objets mathématiques compréhensibles et plus ou moins facile à interpréter par l'utilisateur, ce qui est le majeur avantage et atout de cette méthode.

L'idée de résoudre des problèmes de régression symbolique en algorithmie évolutionnaire a été proposée par John KOZA [Koz 1992] et est, depuis, devenue un domaine largement étudié.

Parmi les problèmes que l'on peut résoudre avec la régression symbolique, citons « la découverte d'identités trigonométriques » qui tente de découvrir de nouvelles expressions trigonométriques. Si on dispose, par exemple de la fonction $\cos(2x)$ de manière symbolique (section 3.1 du chapitre 1), on peut essayer de trouver une forme symbolique équivalente, pour un certain nombre de valeurs x_i dans un intervalle donné : une expression mathématique telle que $1-2\times\sin^2(x)$ peut être une nouvelle identité.

Un autre problème classique est le problème « d'intégration symbolique » [Mitchell 83] où on essaie de trouver la forme symbolique de l'intégrale d'une fonction, la solution peut être parfaite ou une approximation de la fonction d'intégration. Comme dans l'exemple précédent, on dispose d'un certain nombres de paires de points (x_i, y_i) de la courbe dans un certain intervalle et on essaie de trouver l'intégrale qui correspond à cette courbe. Par exemple, si notre courbe est celle de la fonction : $3x+\sin(x)$

on cherchera à trouver comme solution sous forme symbolique la fonction : $2/3x^2-\cos(x)+1$

Le même principe peut se transposer aux problèmes de « *dérivation symbolique* », dans lesquels on essaie de trouver la dérivée d'une fonction. Il existe de nombreux autres problèmes en régression symbolique, tel que trouver les racines numériques d'une équation, construire un modèle à partir des données bruitées d'un système et autres [koz 1992].

Nous allons étudier à l'aide de la régression symbolique les effets des terminaux pertinents sur l'évolution en programmation génétique. Koza [koza 1992] y a consacré un chapitre et conclut que les terminaux extérieurs dégradent la performance de la PG.

Dans le chapitre 1, nous avons vu les étapes préparatoires à l'exécution d'un « run » de programmation génétique (choix de l'ensemble des terminaux et de fonctions, définition de la fonction objective, choix des paramètres internes de la PG, critère d'arrêt et détermination d'une solution au problème). Nous allons voir l'application de ce paradigme à des problèmes de régression symbolique à l'aide de la notion de

sélection d'attributs. Comme on l'a vu précédemment la performance de la programmation génétique dépend du choix des ensembles terminaux et fonctions. S'il n'y a pas assez de fonctions ou de terminaux pour exprimer la solution, on n'atteint pas la forme du problème souhaité, et s'il y a plus de terminaux et fonctions que nécessaire, la taille de l'espace de recherche risque de devenir trop importante par rapport au problème à résoudre.

Dans ce chapitre, nous allons déterminer dans le cadre général de la régression symbolique de polynômes l'impact de la présence de terminaux non pertinents, et nous étudierons l'amélioration potentielle de la performance de la PG à l'aide de la sélection d'attributs pour réduire l'impact des terminaux non pertinents.

Dans notre cas, à partir d'un certain nombre M, cardinal de notre ensemble d'attributs, on tentera de trouver N attributs pertinents, sachant que $N \leq M$ Nous proposerons une méthode pour éliminer les attributs les moins pertinents.

3.2 Sélection de terminaux

Nous nous intéressons à l'impact des terminaux au cours de la PG, et pour cela nous nous appuyons sur la mesure du poids de l'ensemble des attributs.

3.2.1 Mesure de poids des attributs

Nous avons vu dans le chapitre précédent la notion de sélection d'attributs, qui dans notre contexte va nous aider à réduire le nombre de terminaux possibles. Nous allons par la suite utiliser la sélection d'attributs afin de déterminer les terminaux pertinents parmi ceux de l'ensemble des terminaux.

Nous nous proposons d'utiliser une formule de mesure de poids des attributs (« *weithing method* ») inspirée de la sélection d'attributs ; méthode introduite au chapitre 2 qui s'intéresse à l'évaluation individuelle des attributs. Cette technique nous aidera à évaluer les attributs à chaque génération et leur affecter une métrique qui nous donnera un indice sur le degré de pertinence des attributs. Notre méthode est basée sur la fréquence d'apparition des terminaux dans les programmes les mieux classés selon la fonction objectif.

Notre formule de mesure du poids des attributs inspirée de la formule utilisée par Ok [ok 2000] est la suivante:

$$p_n(g) = \sum_{i \in S} (fit_i(g) \times freq_{i,n}(g)) \qquad (1)$$

où,

$p_n(g)$: le poids d'un terminal n à la génération g,

$fit_i(g)$: est la valeur de la fitness du programme i à la génération g,

$freq_{i,n}(g)$: est la fréquence d'apparition du terminal n dans un programme i à la génération g,

S : est l'ensemble des meilleurs 10% des programmes d'une génération selon leur fitness.

Le poids d'un terminal x_i à la génération t va être déterminé en fonction du nombre d'apparitions de ce terminal dans les 10% des meilleurs individus de la génération t et en fonction du fitness de ses individus. Ce qui peut nous donner un aperçu *a priori* sur la pertinence d'un terminal, car plus on a de probabilité de trouver un terminal pertinent dans l'arbre des meilleurs programmes de la génération, et plus les terminaux apparaissent dans les individus ayant les meilleurs fitness.

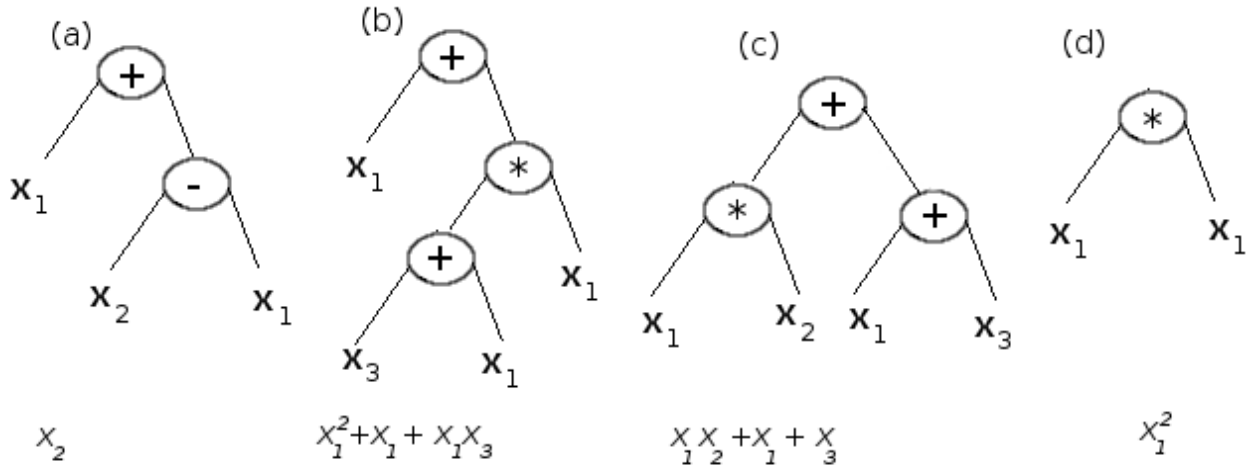

Figure 14: Population de quatre programmes

Supposons que nous effectuons la régression d'une fonction $f(x_1, x_2, x_3)=x_1^2+x_1$, avec une population de 4 individus (figure 14). Afin de calculer le poids des terminaux $x_1, x_2, x_3.$ il nous faut la fréquence d'apparition des trois terminaux dans chaque programme, ainsi que la fitness des programmes (La somme des valeurs absolues des erreurs sur les points d'apprentissage).

Pour
$(x_{1,}x_{2,}x_3; f(x_{1,}x_{2,}x_3))\in\{(-1,-1,-1;0),(0.3,0,1;O.39),(1,-0.9,-0.1;2),(0.5,0.5,0.9;0.75)\}$
,

On a la fitness du programme *a* qui est 4,54, 1,85 pour *b*, 4,08 pour *c* et 2,8 pour *d*. on prend dans cet exemple 50% des individus avec les meilleurs fitness, soit *b* et *d*. ainsi les poids des terminaux dans cette génération g , sont:

$$p_1(g) = 1{,}85 * 3 + 2{,}08 * 2 = 11{,}15$$
$$p_2(g) = 1{,}85 * 0 + 2{,}08 * 0 = 0$$
$$p_3(g) = 1{,}85 * 1 + 2{,}08 * 0 = 1{,}85$$

Nous allons ainsi utiliser cette mesure pour limiter l'impact des terminaux non pertinents.

En effet, l'opérateur évolutionnaire de mutation choisit un point au hasard dans l'arbre d'un programme, ce point peut être un nœud fonction ou un nœud terminal. L'opérateur normal de mutation va l'éliminer ainsi que le sous-arbre dont il est racine (dans le cas d'un nœud fonction), pour le remplacer par la suite par un nouveau sous-arbre généré aléatoirement. Nous nous sommes basés sur la proposition de Ok [Ok 2000] pour les travaux que nous avons effectués. En effet, il a proposé un opérateur nommé « *adaptive mutation* » qu'on peut résumer en deux étapes :

1. Les terminaux sont regroupés en deux catégories : ceux supposés pertinents et les supposés non pertinents, selon la mesure de poids des terminaux. Ses deux groupes seront mis à jour suivant les générations jusqu'à ce qu'on ait assez d'informations pour juger de la pertinence des terminaux.
2. Processus de mutation : quand les terminaux pertinents sont découverts, tous les terminaux non pertinents seront éliminés de tous les individus. Avant de découvrir les terminaux pertinents, l'opérateur de mutation choisit avec une plus grande probabilité des terminaux ayant des poids supérieurs pour construire les nouveaux sous-arbres.

Nous avons fait le choix de considérer que les terminaux ayant un poids normalisé supérieur à 0.8 ont plus de chance d'être des terminaux pertinents et ceux ayant un poids normalisé inférieur à 0.1 ont moins de chance d'être des terminaux pertinents et plus de chance d'être non pertinents. C'est dans cette perspective que nous avons modifié l'opérateur standard de mutation de manière à ne pas inclure dans les nouveaux arbres des terminaux ayant un poids normalisé inférieur à 0.1. Ils sont remplacés au hasard par un terminal ayant un poids normalisé supérieur à 0.8. Nous espérons que, avec ces différentes modifications de la fonction de poids qui ne prend en considération que les performances des terminaux par génération, permettront de distinguer rapidement les terminaux non pertinents de ceux pertinents. La figure 15 montre le processus de notre opérateur de mutation.

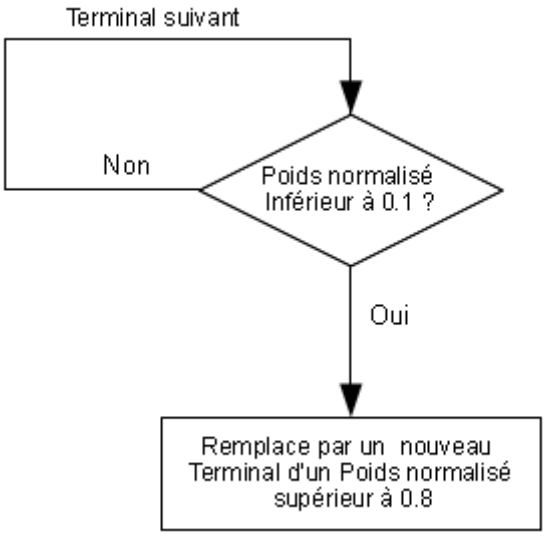

Figure 15: mutation biaisée

3.3 Expérimentations

3.3.1 Problème de régression simple.

Notre première fonction à approcher est la fonction $f_1(x_{1,}x_{2,}\dots,x_{33})=x_1^3+x_1^2+x_1$ où les valeurs des variables sont comprises dans l'intervalle [-1, 1].

Les deux premières étapes de construction de l'algorithme consistent à fournir les ingrédients au processus évolutionnaire qui va permettre de construire une solution au problème. L'ensemble des terminaux est volontairement plus étendu que nécessaire et contient 33 variables d'entrée du programme de x_1 à x_{33} , sachant que l'on aura besoin que d'une seul d'entre elles pour avoir résolu le problème totalement.

$$T=\{x_{1,}x_{2,}\dots,x_{33}\}$$

On utilisera pour l'ensemble des fonctions, en plus des simples fonctions arithmétiques addition, multiplication, soustraction et division, les fonctions trigonométriques sinus et cosinus ainsi que la fonction logarithme.

$$F=\{+,-,*,/,\cos(),\sin(),\log()\}$$

Dans le cadre de la PG, comme expliqué au chapitre 1, la division et la fonction logarithme sont des fonctions protégées, qui testent leur entrée avant l'exécution afin de conserver l'importante propriété de clôture de la programmation génétique.

La construction de la fonction objective est évidente pour ce type de problème : trouver un programme qui a pour sortie les mêmes valeurs que notre polynôme $x_1^3+x_1^2+x_1$. Le fitness est défini par l'intégrale de la différence entre la fonction générée par la PG et la fonction qu'on cherche à régresser. Comme il n'est pas aisé de calculer cette intégrale de façon analytique, on utilise généralement la somme des erreurs en valeur absolue (*sums of the absolute errors)* mesurée selon les différentes valeurs d'entrées x_i de l'intervalle [-1, 1]. Le meilleur fitness serait une valeur qui s'approche de 0, et si elle égale 0 alors les deux fonctions ont la même image pour toute valeur x_i dans le cadre de ces problèmes de régression symbolique, nous choisissons de tester la fitness avec 80 valeurs aléatoires dans l'intervalle [-1, 1].

Parmi les autres paramètres à fixer, nous avons choisi une population de 450 programmes (*cf* tableau 1).

Nous avons testé le problème 30 fois avec différentes graines. Quatre expérimentations ont été effectuées :

- la première expérience consiste à faire une régression symbolique standard du problème, mais sans la présence de terminaux non pertinents, ce qui revient au problème de régression avec un seul terminal x ;
- la deuxième expérience consiste à utiliser une régression symbolique standard avec 33 terminaux dont un seul est pertinent ;
- la troisième expérience utilise notre méthode de mutation qui s'appuie sur la mesure de poids ;
- la quatrième expérience utilise le même principe que l'expérience précédente, mais en utilisant cette fois la formule de poids proposée par [ok 2000] (*cf infra*).

Notons que nous avons implémenté ces différents problèmes en utilisant la librairie ECJ (http://cs.gmu.edu/~eclab/projects/ecj/).

Objectif	Trouver une représentation symbolique de la fonction fonction qui corresponde aux points données
Ensemble des terminaux	$x_1, x_2, \ldots x_{33}$
Fonction objectif	La somme en valeur absolue des erreurs entre les valeurs de la fonction cible $y(i)=x_1^3(i)+x_1^2(i)+x_1(i)$ et le programme en 80 points $(x_1(i),\ldots,x_{33}(i))$ de l'intervalle $[-1,1]^{33}$.
Hits	Nombre de fois la valeur où y(i) est égal(ou avec une différence inférieure à 0,01) à la valeur produite par la fonction $y(i)=x_1^3(i)+x_1^2(i)+x_1(i)$
Paramètres	Taille de la population =450, génération=100,
Croisement	80%
Reproduction	10%
Mutation	10%
Critère d'arrêt	L'expression obtient 80 Hits

Tableau 1: Tableau des paramètres de la PG

La figure 16 décrit le taux de succès cumulé *P(m,i)*, mesure statistique définie par Koza [Koz 1992] qui détermine la probabilité cumulée de trouver un individu idéal pour ce problème à la génération i, où m est la taille de la population.

On remarque également qu'avec notre méthode de mutation où on utilise notre mutation basée sur la discrimination des terminaux, on améliore le taux de succès de la PG avec terminaux non pertinents et on s'approche du taux de sucées de la PG en l'absence des terminaux non pertinents. La comparaison entre la PG standard avec terminaux non pertinents et la PG avec notre mutation montre un gain important pour cette dernière.

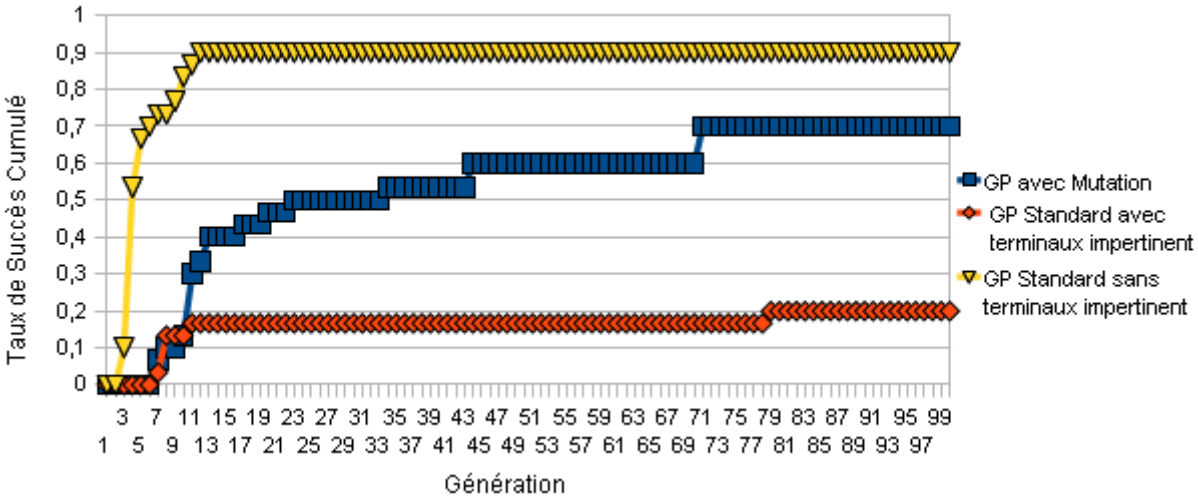

Figure 16: Taux de succès cumulé du problème de régression des trois types de PG

Le quatrième type d'expérience consiste comme indiqué précédemment à utiliser notre opérateur de mutation (Figure 1) avec une fonction de poids qui tient compte des mesures du poids dans les générations précédentes et cumule les poids génération par génération. On ajoute ainsi le poids du terminal de la génération précédente au poids de la génération actuelle. La formule (2) suivante est présentée ci-dessous prend en compte ce cumul de poids :

$$p_n(g)=\sum_{i\in S}(fit_i(g)\times freq_{i,n}(g))+p_n(g-1) \qquad (2)$$

La figure 17 nous montre la comparaison entre le taux de succès cumulé des deux méthodes, avec et sans cumul de poids. On peut remarquer que la méthode avec cumul de poids tarde à trouver des solutions de bonne qualité contrairement à celle sans cumul qui a un taux de succès largement supérieur.

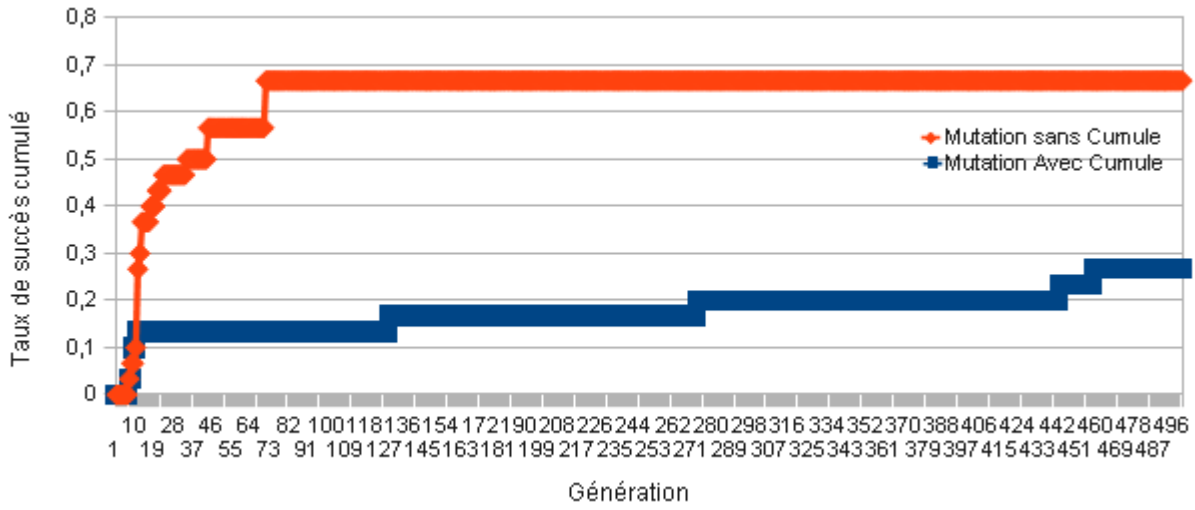

Figure 17: Évolution du taux de succès cumulé entre les deux formules de mesure de poids.

La figure 18 étudie l'évolution du poids du terminal pertinent x1 et la moyenne des 32 terminaux non-pertinents sur les deux techniques. La figure 17 nous montre qu'il y a une différence importante entre les deux méthodes. En effet, dès le départ le poids normalisé du terminal pertinent x1 tend vers 1, ceci est dû à la chute rapide des poids normalisés des terminaux non pertinents et leur passage au dessous de la valeur 0,1, leurs poids se stabilisant très rapidement autour de la valeur 0. Les poids normalisés qui utilisent la fonction de mesure de poids avec cumul tardent à mettre les terminaux non pertinents au-dessous de 0,1. En examinant de manière conjointe les figures 17 et 18, on peut remarquer que dès que la moyenne des terminaux non pertinents passe au-dessous de 0,1, le taux de succès cumulé de la PG augmente de manière significative.

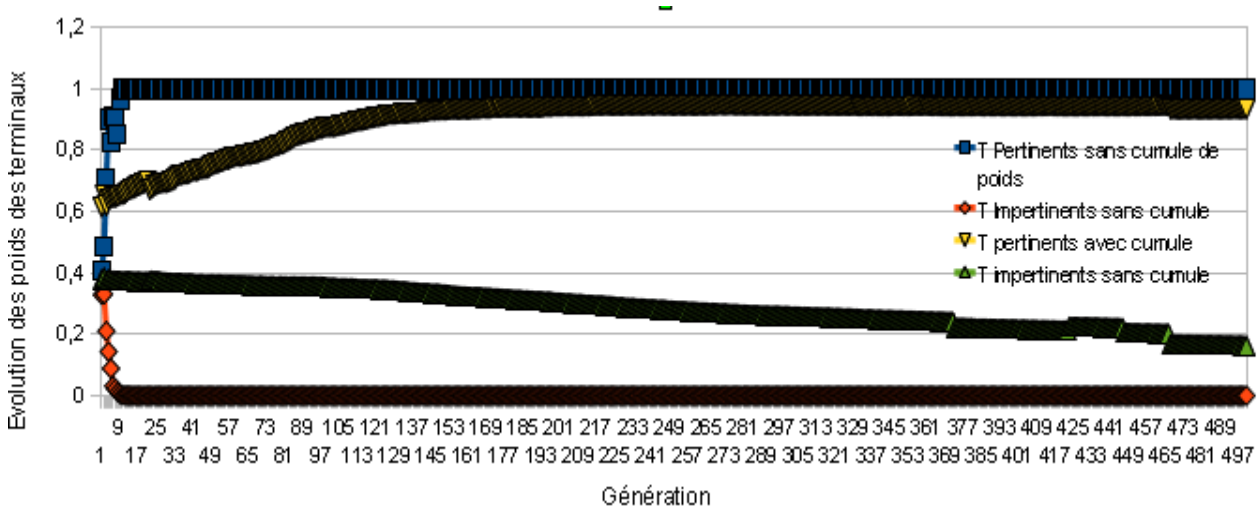

Figure 18: L'évolution du Poids normalisé des terminaux pertinents (x1) et la moyenne des terminaux non-pertinents dans les expériences 3 et 4

3.3.2 Problèmes complexe de régression symbolique.

Le deuxième problème de régression symbolique proposé par [ok 2000] est plus complexe et avec plus de terminaux pertinents,

$$f_2(x_{1,..},x_{33})=\sin(x_4)+\sin(2*x_4)+\sin(3*x_4)+\sin(4*x_4)+\sin(x_5)+\sin(2*x_5)+\sin(3*x_5)$$
$$+\sin(x_{13})+\sin(2*x_{13})$$

Vu la difficulté du problème, on utilise 2000 individus et 500 générations.

Le tableau 2 donne les résultats des fitness du meilleur individu et la moyenne sur 10 expériences des quatre types de PG introduits précédemment. La comparaison des moyennes de fitness brute nous montre que les PG avec mutation biaisée sont meilleures que la PG standard avec terminaux non pertinents, et s'approchent du résultat obtenu par la PG standard sans terminaux non pertinents. Notons également que le quatrième opérateur a eu légèrement de meilleurs résultats que le troisième opérateur.

	Fitness		
	Fitness Brute (Raw[1] fitness)	Fitness ajusté (adjusted fitness[2])	Hits
PG standard sans terminaux non-pertinents	Meilleur = 2,46 Moyenne =9,46	0,28 0,11	11/80 8,5/80
PG standard, l'ensemble des terminaux contient les terminaux non-pertinents	Meilleur = 3.55 Moyenne = 12,49	0.21 0,09	17/80 7,1/80
PG où l'opérateur de mutation utilise la formule 1 (sans cumul de poids)	Meilleur = 3,60 Moyenne = 9,09	0,219 0,104	16/80 8,6/80
PG où l'opérateur de mutation utilise la formule 2 (avec cumul de poids)	Meilleur = 2.01 Moyenne = 9,74	0.33 0,105	32/80 8,5/80

Tableau 2: les résultat des 4 types de GP du deuxième problème avec le problème f_2 .

Les figures 19a et 19b montrent l'évolution du poids normalisé des terminaux pertinent et non pertinent dans les deux types de PG avec mutation (19a mesure de poids sans cumul). On remarque que dès les premières générations (figure 19a) les terminaux non-pertinents sont passés au-dessous de ***0.1*** ainsi que le terminal pertinent x_4 qui est au-dessus de ***0.8*** et y reste. x_5 et x_{13} sont rapidement détectés même si leur poids n'atteint pas celui de x_4 . Pour la PG avec mesure de poids avec cumul (figure 19b), les terminaux non pertinents ont besoin d'un nombre de générations plus important pour passer au-dessous de la barre des ***0.1***. L'évolution des courbes des terminaux pertinents est quasiment similaire vers la fin du programme même si le terminal x_4 a besoin de plus de générations pour voir son poids normalisé s'approcher de ***0.1***.

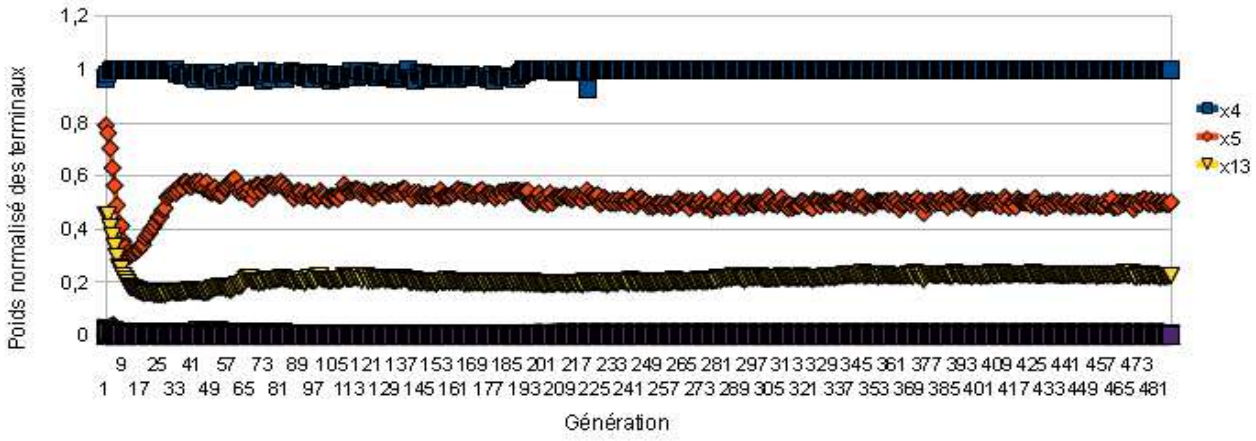

Figure 19a: Évolution des poids normalisé des 33 terminaux dans la PG où la mutation utilise la fonction de mesure de poids sans cumul, où seuls les terminaux x4,x5 et x13 (pertinents) ont un poids normalisé supérieur à 0,1. Les poids des 30 terminaux non-pertinents tendent vers O dès les premières générations .

1 Raw fitness est la fitness brute qui représente la capacité du programme à trouvé une solution au problème dans la terminologie de ce problème [Koza 1992], la meilleur valeur de fitness brute est 0.

2 Adjusted fitness ou la fitness ajusté réduit l'ntervalle des valeurs à [0, 1]

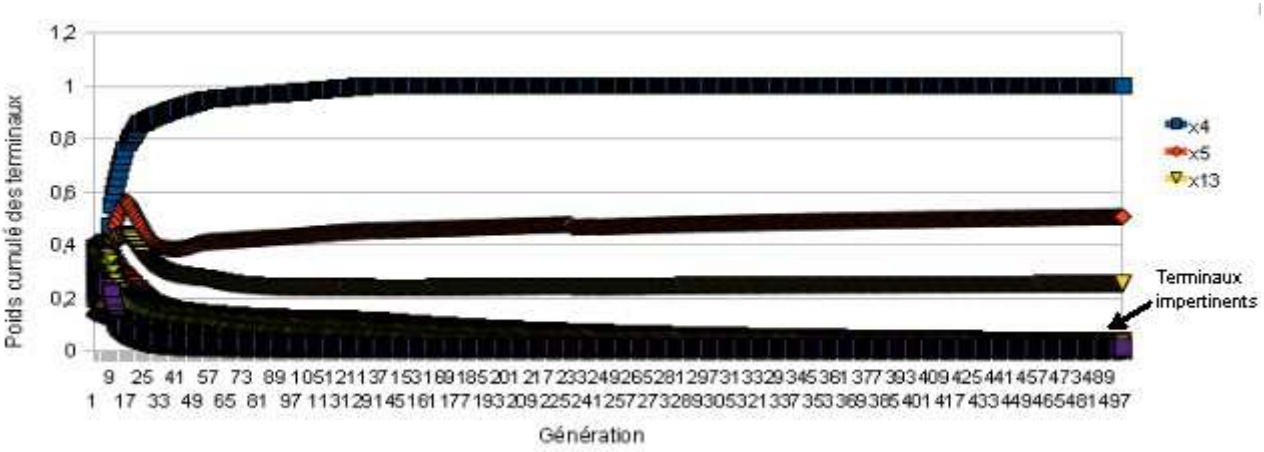

Figure 19b: Évolution des poids normalisés des 33 terminaux dans la PG en utilisant la fonction de mutation avec cumul du poids.

On remarque que dès les premières générations les deux approches de mutation se différencient ; les poids normalisés des terminaux non pertinents sans cumul passent très rapidement sous la barre de ***0.1***, ce qui se répercute sur l'évolution des terminaux pertinents. Contrairement au poids avec cumul où les terminaux non-pertinents tardent à passer au-dessus de ***0.1***, ce qui explique le léger avantage de la méthode sans cumul sur celle avec cumul. Cependant, les deux approches obtiennent des résultats similaires.

3.3.1 Troisième problème de régression symbolique

Le troisième problème est une fonction constituée de 17 terminaux parmi l'ensemble des 33 terminaux.

$$f_3(i)=x_1(i)+x_3(i)+x_5(i)+x_7(i)+x_9(i)+x_{11}(i)+x_{13}(i)+x_{15}(i)+x_{17}(i)+x_{19}(i)+x_{21}(i)+x_{23}(i)$$
$$+x_{25}(i)+x_{27}(i)+x_{29}(i)+x_{31}(i)+x_{33}(i)$$

Les paramètres d'exécution du deuxième problème sont conservés pour ce problème avec 16 terminaux non pertinents.

Le tableau 3 montre les résultats obtenus par les quatre types d'expérience. Les deux variantes n'améliorent pas les performances de la PG standard avec terminaux non pertinents, mais on remarque néanmoins que la version sans cumul de poids est meilleure que la mutation avec cumul des poids des terminaux sur ce problème.

	Fitness		
	Fitness Brute (Raw fitnes)	Fitness ajusté (adjusted fitness)	Hits
PG standard sans terminaux non-pertinents	Meilleur = 0	1,0	80/80
	Moyenne =7,04	0,80	64,6/80
PG standard, l'ensemble des terminaux contient les terminaux non-pertinents	Meilleur = 0	1,0	80/80
	Moyenne = 9,67	0,61	49,5/80
PG où L'opérateur de mutation utilise la formule 1 (sans cumul de poids)	Meilleur = 0	1	80/80
	Moyenne = 24,51	0,31	36,6/80
PG où L'opérateur de mutation utilise la formule 2 (avec cumul de poids)	Meilleur = 26,90	0,03	80/80
	Moyenne = 46,1	0,02	26,6/80

Tableau 3: les résultat des 4 types de GP du deuxième problème avec le problème f_3 .

Les figures 20 et 21 montrent les graphes des deux approches de mutation où on affiche séparément l'évolution des poids normalisés des terminaux pertinents et non pertinents. On remarque pour la figure 20 que la chute précoce des poids des terminaux non pertinents n'a pas de bonnes répercussions sur la performance de cet opérateur de mutation.

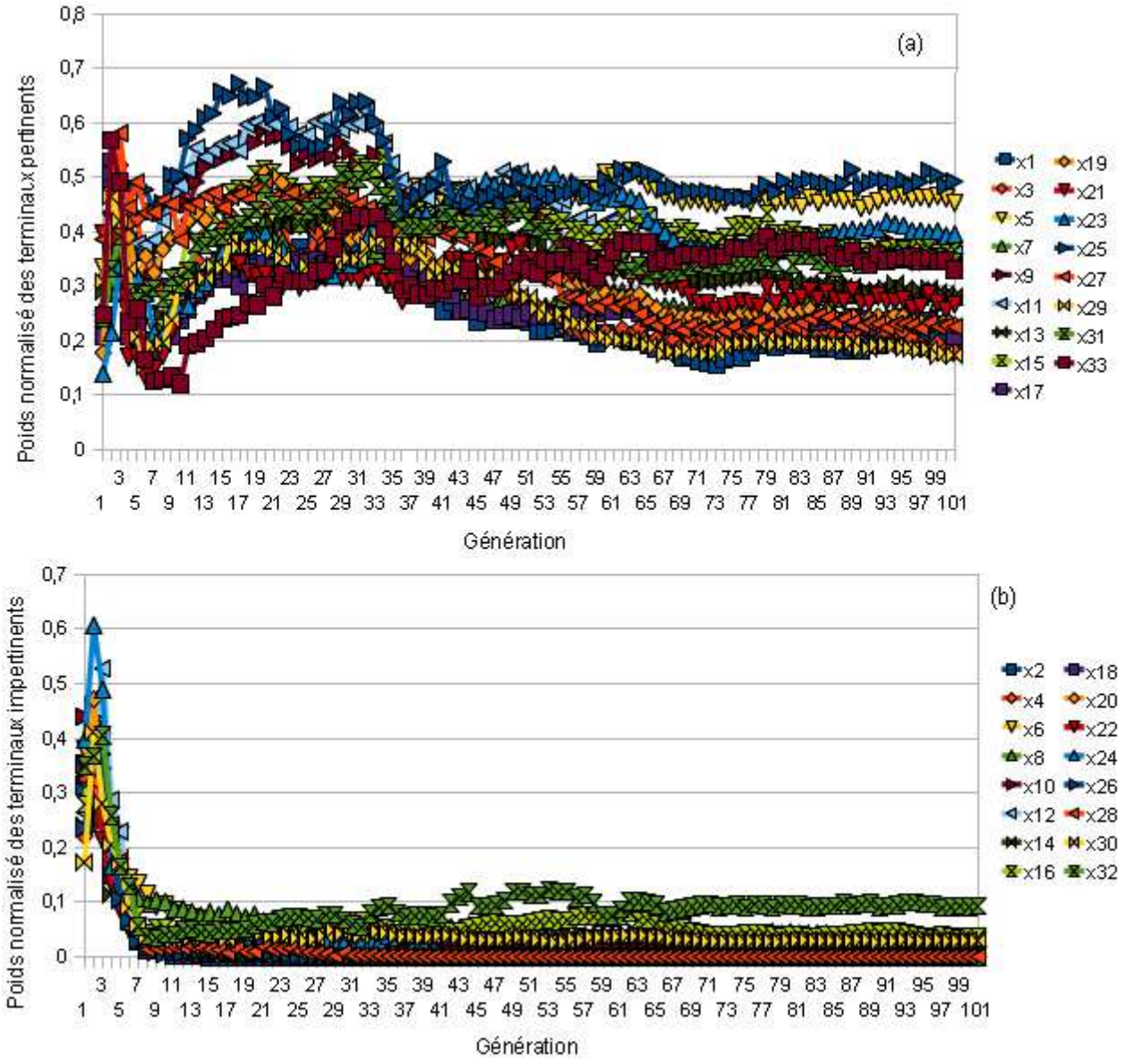

Figure 20: évolution des poids normalisés des terminaux pertinents (a), et impertinents (b). (Mesure de Poids sans cumul)

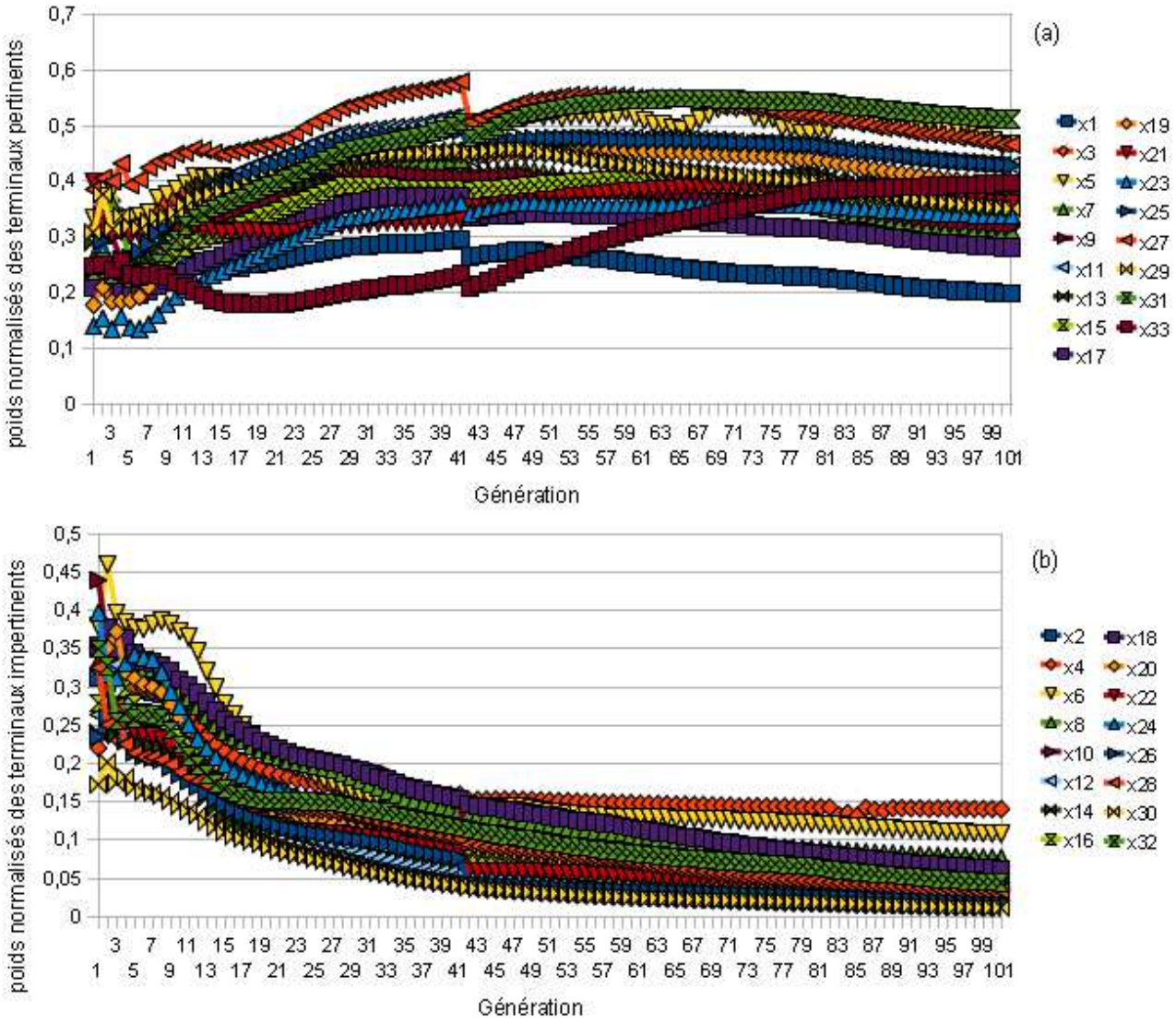

Figure 21: 'évolution des poids normalisés pertinents (a), et impertinents (b). (Mesure de Poids avec cumul)

3.4 Conclusion

Nous avons expérimenté deux approches d'un opérateur de mutation qui utilisent la notion de mesure de poids de la sélection d'attributs dans le but de réduire l'impact négatif des terminaux non pertinents sur la performance du PG. On peut remarquer que la formule que nous proposons (sans cumul) obtient généralement de meilleures performances que la variante qui utilise la mesure de poids avec cumul des poids des terminaux au fil des générations, et s'approche généralement des performances de la PG sans les terminaux non pertinents.

Il faudrait étudier sur d'autres types de problèmes pourquoi ces deux méthodes n'obtiennent pas de bons résultats sur la troisième expérience tout en étant capables de discriminer les terminaux pertinents des terminaux non pertinents.

4 Chapitre 4 : Utilisation de l'évolution différentielle pour la programmation génétique

4.1 Introduction

Ce chapitre se situe quelque peu en dehors des travaux présentés dans le cadre de ce travail de thèse. En effet, il se situe dans le cadre général de l'amélioration de la PG par des techniques connexes aux algorithmes évolutionnaires. En effet, la plupart des « moteurs » (au sens du mot anglais *engine*) sont directement calqués sur l'utilisation des algorithmes génétiques. La boucle évolutionnaire dans le cadre de la PG consiste en plusieurs phases :

- initialisation des programmes ;
- sélection des programmes ;
- application d'opérateurs évolutionnaires sur les programmes ;
- insertion éventuelle des nouveaux programmes.

Nous avons choisi dans le cadre de ce chapitre de nous inspirer d'une méthode d'optimisation stochastique continue très performante, l'évolution différentielle (« Differential Evolution ») qui nous servira de base à la construction de notre moteur pour notre boucle évolutionnaire. Pour ce faire, nous avons décidé non pas de manipuler des arbres classiques (à la Koza) mais de directement manipuler des programmes linéaires (cadre de la programmation génétique linéaire).

Dans ce chapitre, nous présenterons successivement les bases de l'Évolution Différentielle (DE), de la programmation génétique linéaire. Puis, nous présenterons l'adaptation de la PG à l'évolution différentielle et nous présenterons enfin un certain nombre de résultats expérimentaux.

4.2 Les concepts de l'évolution différentielle

L'évolution différentielle que nous noterons DE pour des raisons de simplicité est une technique d'optimisation continue proposée par Storn et Price en 1997 [Sto 1997]. Comme dans le cadre des stratégies d'évolution, DE utilise des informations extraites de la population courante afin de déterminer l'évolution future de la recherche.

Dans cette section, nous ne présentons que les caractéristiques essentielles de DE ; des détails complets et une présentation exhaustive de DE est disponible dans [Sto 1997].

L'évolution différentielle travaille sur un ensemble de N vecteurs de dimension d X_i qui sont appelés « vecteurs solutions », chaque élément du vecteur correspondant à une valeur réelle.

$$X_i=(x_{i1},x_{i2},\ldots,x_{id}) \quad i=1,2,\ldots,N \quad (1)$$

DE s'inspire en partie de la boucle évolutionnaire car il peut se décomposer en plusieurs phases proches des techniques évolutionnaires :

- initialisation ;
- mutation ;
- croisement ;
- évaluation ;
- sélection.

Initialisation : comme DE est une méthode d'optimisation **continue**, la population de vecteurs solutions (*cf 1*) est en général initialisée par une loi gaussienne ;

mutation : la mutation est une phase importante pour DE car elle permet la génération d'un vecteur variant qui est généré pour chaque vecteur X_j de la population. Ce vecteur est une perturbation du vecteur X_i à partir de trois autres vecteurs de la population. De manière formelle :

$$V_j(t+1)=X_{r_1}(t)+F\times(X_{r_2}(t)-X_{r_3}(t)) \quad (2)$$

où r_1, r_2, et r_3 sont indices de vecteurs de la population sélectionnés aléatoirement et différents les uns des autres et différents de j. F est une constante réelle qui permet de contrôler l'amplification de l'évolution différentielle et évite ainsi la stagnation au cours de la recherche. Il est admis de prendre une valeur de F dans l'intervalle [0,2].

t est le numéro de la génération courante. Il existe de nombreuses variantes concernant l'équation 2 qui permettent de faire intervenir 2, 3 ou 4 vecteurs différents.

D'après [Veen09,Pric96], la formule de mutation qui délivre les meilleurs résultats est la méthode appelée « DE/best/2/bin », qui est définie par l'équation suivante :

$$V_j(t+1)=X_{best}(t)+F\times(X_{r_1}(t)+X_{r_2}(t)-X_{r_3}(t)-X_{r_4}(t)) \quad (3)$$

$X_{best}(t)$ étant le meilleur individu dans la population à la génération t. Dans le cadre de ce travail de thèse, nous avons choisi cette méthode de mutation.

Croisement : le croisement permet dans le cadre de l'évolution différentielle de conserver un certain taux de diversité dans la population. Dans cette étape de la boucle évolutionnaire, un vecteur d'essai est créé et l'opérateur de croisement garantit qu'au minimum un des éléments du vecteur variant sera conservé. Le vecteur est $U_j=(u_{j1},u_{j2},\ldots,u_{jd})$ est créé à partir de la formule suivante :

$$u_{ji}(t+1)=\begin{cases} v_{ji}(t+1) \; si \; rand \leqslant CR \quad ou \quad j=rnbr(i) \\ x_j(t) \; si \; rand > CR \quad ou \quad j \neq rnbr(i) \end{cases} \quad (4)$$

où x_{ji} est le jième composant du vecteur x_i, v_{ji} est le jième composant du vecteur variant $V_j(t+1)$; *rand* est une valeur aléatoire entre [0,1] ; *CR* est le taux de croisement et enfin rnbr(i) est une valeur aléatoire entière comprise entre [1,d], ce qui permet de garantir qu'au moins un des éléments du vecteur variant sera transmis au vecteur d'essai.

Sélection : l'étape de sélection permet de remplacer le vecteur $X_i(t)$ par le vecteur d'essai $U_i(t+1)$ dans la population si sa fitness est meilleure.

Ces quatre étapes sont réitérées jusqu'à ce qu'un nombre maximal d'itérations soit atteint ou jusqu'à ce qu'une solution soit trouvée. Une des grandes force de l'évolution différentielle est qu'elle est relativement simple à programmer et que seuls trois principaux paramètres sont nécessaires à son bon fonctionnement : la taille de la population (N), le taux de croisement (CR) et le facteur d'échelle (F).

Le tableau suivant résume les principales étapes de l'évolution différentielle :

pour tous les vecteurs X_i faire :

calculer r_1, r_2, r_3 et r_4 tels que $i \neq r1 \neq r2 \neq r3 \neq r4$

calculer le vecteur variant V_i en utilisant l'équation (3)

créer le vecteur d'essai U_i en utilisation l'équation (4)

si $f(U_i) < f(X_i)$ remplacer X_i par U_i

fin pour

génération = génération + 1

4.3 La programmation génétique linéaire

Dans ce chapitre, nous introduisons de manière succincte la programmation génétique linéaire qui est une branche du large champ de la PG. Une étude complète sur ce domaine est disponible dans [Bra 2007]. Cette technique de PG se différencie principalement de la PG « classique » à la Koza par le fait que ce ne sont plus des arbres représentant des programmes qui sont manipulés, mais un ensemble de chromosomes formant un code linéaire dans un langage de haut niveau comme le langage C par exemple.

4.3.1 Représentation des programmes

Dans le cadre de la programmation génétique linéaire, un programme consiste en une suite d'instructions séquentielles où chaque instruction est une instruction à trois registres (un registre destination où le résultat de l'instruction sera stocké et deux registres qui seront utilisés comme opérandes). Par exemple, $r_i = r_i + r_j$ est une instruction à 3 registres.

De manière à généraliser notre représentation, les instructions à un seul opérande seront bien évidemment permises et nous autoriserons les constantes comme un des deux opérandes possible. L'exemple ci-dessous présente un exemple typique de programme écrit avec une suite d'instructions à 3 registres :

```
void gp()
{
…
        r[0] = r[3] + r[4] ;
        r[7] = sin(r[1]) ;
        r[4] = r[2] * 0,85 ;
        r[2] = r[7] – r[2] ;
        r[8] = r[3] – 0,25 ;
        r[8] = cos(r[0]) ;
…
}
```

4.3.2 Représentation - codage des instructions

Dans l'implantation que nous avons choisie, un programme sera une suite de valeurs réelles où une instruction sera une suite de quatre valeurs réelles. Par exemple, l'instruction $r_i = r_i + r_j$ sera représentée par le groupe de quatre valeurs <id(+),i,j,k>.

Le programme est donc un tableau de réels dont la taille peut éventuellement varier suivant la taille du programme que l'on souhaite obtenir.

Dans le cadre de notre implantation de la programmation génétique linéaire, le nombre de registres qui seront utilisés comme registre de calculs, d'entrée et de sortie est un paramètre fourni par l'utilisateur. Par défaut, tous les registres seront initialisés à la valeur du ou des paramètres d'entrée et la valeur de sortie sera extraite du registre r[0].

Toutes ces instructions sont exécutées par l'intermédiaire d'une machine virtuelle. Afin de permettre de résoudre des problèmes complexes, la présence de constantes est indispensable. Dans notre cas, les constantes sont générées de manière aléatoire dans un intervalle défini par l'utilisateur et seront accessibles en lecture seulement. La présence des constantes dans le programme est modulée par une constante de probabilité des constantes (p_c).

4.3.3 Implantation

Comme indiqué précédemment, un programme dans le cadre de la PGL consiste simplement en un tableau de valeurs réelles. La PGL découpera ce tableau en tranche de quatre valeurs réelles (v_1, v_2, v_3, v_4) . v_1 représentera l'opérateur, v_2, v_3 et v_4 représenteront les opérandes. Les valeurs réelles sont transformées en valeurs entières, valeurs entières qui serviront d'indices à l'ensemble d'opérateurs et d'opérandes. La formule (5) retourne le numéro de l'opérateur :

$$opérateur = (\lfloor (v_j - \lfloor v_j \rfloor) \times n_{opérateurs} \rfloor) \quad (5)$$

où $n_{opérateurs}$ est le nombre total d'opérateurs.

Dans le cas de décodage d'un opérande, deux cas peuvent apparaître : soit la partie décimale de $(v_j - \lfloor v_j \rfloor)$ est supérieure à la probabilité de constante et on a affaire à un registre, soit la valeur retournée est inférieure et on va récupérer un indice qui indexera le tableau de constantes de taille C générées aléatoirement à l'initialisation.

$$si \ (v_j - \lfloor v_j \rfloor) \leqslant p_c \quad constante = \lfloor v_j \times C \rfloor \ mod \ C \quad (6)$$

$$si \ (v_j - \lfloor v_j \rfloor) \geqslant p_c \quad registre = (\lfloor (v_j - \lfloor v_j \rfloor) \times n_{registres} \rfloor) \quad (7)$$

Exemple :

Supposons que l'ensemble des opérateurs est { 0:+, 1:-, 2:x, 3:/ }, que 6 registres sont utilisés (r_0 à r_5), 50 constantes sont disponibles et que la probabilité d'apparition de constantes est de 0,1. Notre programme se compose de 8 réels soit 2 instructions :

0,17	2,41	1,84	1,07	0,65	1,22	1,22	4,28

Comme indiqué précédemment, la première valeur indiquera l'opérateur de la première instruction. En utilisant l'équation (5), $(0{,}17 \times 4) = 0$ ce qui signifie que l'opérateur sera l'opérateur d'addition.

La seconde valeur de notre vecteur (2,41) sera transformée en un opérande. En utilisant, l'équation 6, $\lfloor (0{,}41 \times 6) \rfloor = 2$, ce qui indique que le registre 2 (r[2]) sera utilisé pour notre première opération, comme registre de destination.

Pour la troisième valeur (1,84), à nouveau un registre sera utilisé car la partie décimale est supérieure à p_c la probabilité d'apparition des constantes. $\lfloor(0{,}84\times 6)\rfloor=5$, r[5] sera utilisé.

Pour la dernière valeur de notre première instruction, le calcul de $(1{,}07-\lfloor 1{,}07\rfloor)=0{,}07$ retourne un résultat inférieur à la probabilité d'apparition des constantes. En accord avec l'équation (7), une constante sera utilisée et le résultat de l'équation nous retournera l'indice de la constante à prendre dans le tableau des constantes. $\lfloor(1{,}07\times 50)\rfloor mod\ 50=3$.

Les quatre premières valeurs de notre génotype se traduisent donc par l'instruction :

r[2] = r[5] + c_3

Supposons que c_3 corresponde à la valeur 1,87. Le processus de traduction continue pour les quatre valeurs suivantes et on obtient le programme :

r[2] = r[5] + c_3

r[1] = r[1] x r[1]

Quelques informations supplémentaires :

Initialisation : durant la phase d'initialisation, 50 valeurs de l'intervalle [-1,0;+1,0] sont générés aléatoirement et seront utilisées comme constantes pour les expériences en régression symbolique.

Itération : dans le cadre de notre modèle de programmation génétique linéaire, nous avons utilisé deux variantes dans la boucle évolutionnaire : la variante générationnelle et la variante « stable ». La variante générationnelle est la variante originelle définie par Storn et Price ; elle consiste à créer de nouveaux individus dans une génération qui remplaceront leurs parents si leur fitness est meilleure. Dans le cadre de la variante « stable », chaque individu est créé est immédiatement inséré dans la nouvelle population si sa fitness est meilleure que celle de ces géniteurs. Cette variante a été utilisée par Veenhuis [Vee 2009].

4.4 Expérimentations

Afin de valider notre approche, nous avons utilisé un certain nombre de « benchmarks »classiques de la littérature traitant de la programmation génétique (4 problèmes de régression symbolique et un problème sur les fourmis artificielles) auxquels nous avons ajouté deux problèmes de régression symbolique nécessitant la présence de constantes. Nous comparerons également notre approche avec l'approche proposée par Veenhuis [Vee 2009] qui a utilisé la technique d'évolution différentielle pour générer des programmes sous forme d'arbres ; cette technique a cependant un défaut important car elle nécessite l'exécution de la technique plusieurs fois afin de

trouver la taille optimale des arbres, ce qui en fait une technique d'usage pratique fort limitée. Dans les tableaux de résultats, cette méthode est nommée **TreeDE**.

Afin de comparer ces résultats avec les techniques classiques de la PG, on a utilisé la librairie ECJ. Pour tous ces problèmes, 40 runs **indépendants** ont été exécutés. Nous avons également calculé le ratio de « hits » qui caractérise une solution parfaite en PG.

- Les paramètres de contrôle standards ont été utilisés pour l'évolution différentielle, à savoir F = 0,5, CR = 0,1. La taille de la population a été définie à 20 et comme présenté dans l'introduction, la méthode dite /best/2/bin a été retenue. Les résultats ont été calculés pour les variantes générationnelle et « stable » ;
- La taille du vecteur (individu) que la boucle de DE manipule a été fixée à 128 pour les problèmes de régression (ce qui indique une taille de programme de 128/4 = 32 instructions). La taille du programme est réduite à 50 soit 12 instructions dans le cas du problème des fourmis artificielles car ce problème ne nécessite que du code à une seule instruction ;
- La probabilité d'apparition de constantes dans le cadre des problèmes de régression symbolique est fixée à 0,05 ;
- 6 registres ont été utilisés dans les problèmes de régression symbolique (de r[0] à r[5]), r[0] est utilisé comme registre résultat. Tous ces registres sont initialisés au début du programme à la valeur x_k (x_k,y_k) étant un des couples utilisé pour la régression symbolique ;
- dans le cadre de l'évolution différentielle, pour une population de 20 individus, nous avons fixé le critère d'arrêt de la boucle évolutionnaire à 50000 évaluations ;
- pour le problème des fourmis artificielles, problème plus complexe, nous avons fixé le nombre d'itérations maximal à 200000 ;
- les paramètres du programme de PG ont été fixés à des valeurs « classiques »:50 générations et 1000 individus pour les problèmes de régression, et 50 générations et 4000 individus pour le problème de fourmi artificielle. De même, le taux de croisement a été fixé à 80%, 10% pour le taux de mutation et 10% pour le taux de copie. La profondeur maximale admissible pour un arbre est de 11 et nous avons choisi un politique élitiste qui conserve le meilleur individu d'une génération à l'autre.

4.4.1 Problèmes de régression symbolique

Pour les 6 problèmes de régression symbolique que nous présentons dans ce chapitre, le but est de trouver une expression mathématique qui associe 20 points x_k de l'intervalle [-1,0:+1,0] à leur valeur de sortie y_k. Les points d'entrées sont générées aléatoirement

et leurs valeurs de sortie dont calculées en utilisant les fonctions suivantes (de [Vee 2009]) :

$$f_1 = x^3 + x^2 + x$$
$$f_2 = x^4 + x^3 + x^2 + x$$
$$f_3 = x^5 + x^4 + x^3 + x^2 + x$$
$$f_4 = x^5 - 2x^3 + x$$

Ces quatre « benchmarks » très courants dans le cadre de la PG ont été résolus par [Veen09] sans utiliser de constantes. Nous avons utilisé notre technique basée sur l'évolution différentielle sans constante pour avoir une base de comparaison avec des travaux antérieurs mais également avec constantes car nous pensons que des problèmes de régression symbolique nécessitent la présence de constantes, puisqu'il n'est pas possible de supposer la présence ou l'absence de constantes.

Pour les raisons invoquées ci-dessus, nous avons ajouté 2 benchmarks supplémentaires :

$$f_5 = \pi$$
$$f_6 = \frac{x}{\pi} + \frac{x^2}{\pi} + 2x\pi$$

L'ensemble des opérateurs pour ces problèmes est {+, -, x, /} où / est la division protégée.

Le calcul de la fitness est calculé de manière classique comme la somme des différences en valeur absolue sur l'ensemble des points : $\sum |f(x_k) - P(x_k)|$ où P est le programme évolué et k le nombre de couple s (x_k, y_k) (20 dans nos 6 problèmes). Un « hit » est obtenu si un point y_k est obtenu avec une précision inférieure à 10^{-4}.

Le tableau suivant donne les résultats sur les quatre premiers problèmes sans utilisation de constantes :

	DE générationnelle			DE stable			TreeDE			PG		
Problème	Fit	%hits	Eval	Fit.	% hits	Eval.	Fit.	% hits	Eval.	Fit.	% hits	Eval.
f1	0	100	4297	0,0	100	2632	0,0	100	1040	0,0	100	1815
f2	0,0	100	12033	0,0	100	7672	0,0	100	3000	0,0	100	2865
f3	0,28	72,5	21268	0,08	85	21826	0,027	98	8440	0,03	97	6390
f4	0,20	62,5	33233	0,13	75	26998	0,165	68	14600	0,01	80	10845

Il apparaît que pour l'ensemble des problèmes les trois heuristiques ont une comportement relativement proche même si la PG « classique » obtient en moyenne les meilleurs résultats. Il semble que la variante « stable » pour l'évolution différentielle est un comportement meilleur dans le sens où le nombre de générations nécessaires à la convergence est en général plus faible.

Le tableau suivant donne cette fois-ci les résultats sur les 6 problèmes de régression symbolique en introduisant cette fois la présence et la manipulation de constantes :

	DE générationnelle			DE stable			PG		
Problème	Fit	%hits	Eval	Fit.	% hits	Eval.	Fit.	% hits	Eval.
f1	0,0	100	7957	0,0	100	7355	0,002	98	3435
f2	0,02	95	16282	0,0	100	14815	0,0	100	4005
f3	0,4	52,5	24767	0,0	100	10527	0,02	93	7695
f4	0,36	42,5	21941	0,278	45	26501	0,33	23	24465
f5	0,13	2,5	34820	0,06	15	29200	0,07	0	NA
f6	0,59	0	NA	0,63	0	NA	0,21	0	NA

Les résultats de ce tableau confirment que la variante « stable » se comporte mieux que la technique générationnelle. En outre, avec cette technique l'évolution différentielle se compare de manière très favorable par rapport à la PG standard.

4.4.2 Problème de la fourmi artificielle

4.4.2.1 Introduction

Le problème de la fourmi artificielle ou encore la piste de Santa Fe (Santa Fe Tail) est un des problèmes les plus connus en programmation génétique. L'objectif est de construire un programme informatique capable de contrôler à partir de commandes simples une fourmi artificielle qui devra ramasser 89 boulettes de nourriture sur une

piste discontinue, tout cela en un temps limité (400 ou 600 en général), sachant qu'une instruction élémentaire de la fourmi consomme une unité de temps.

La figure 22 présente la piste à parcourir pour la fourmi :

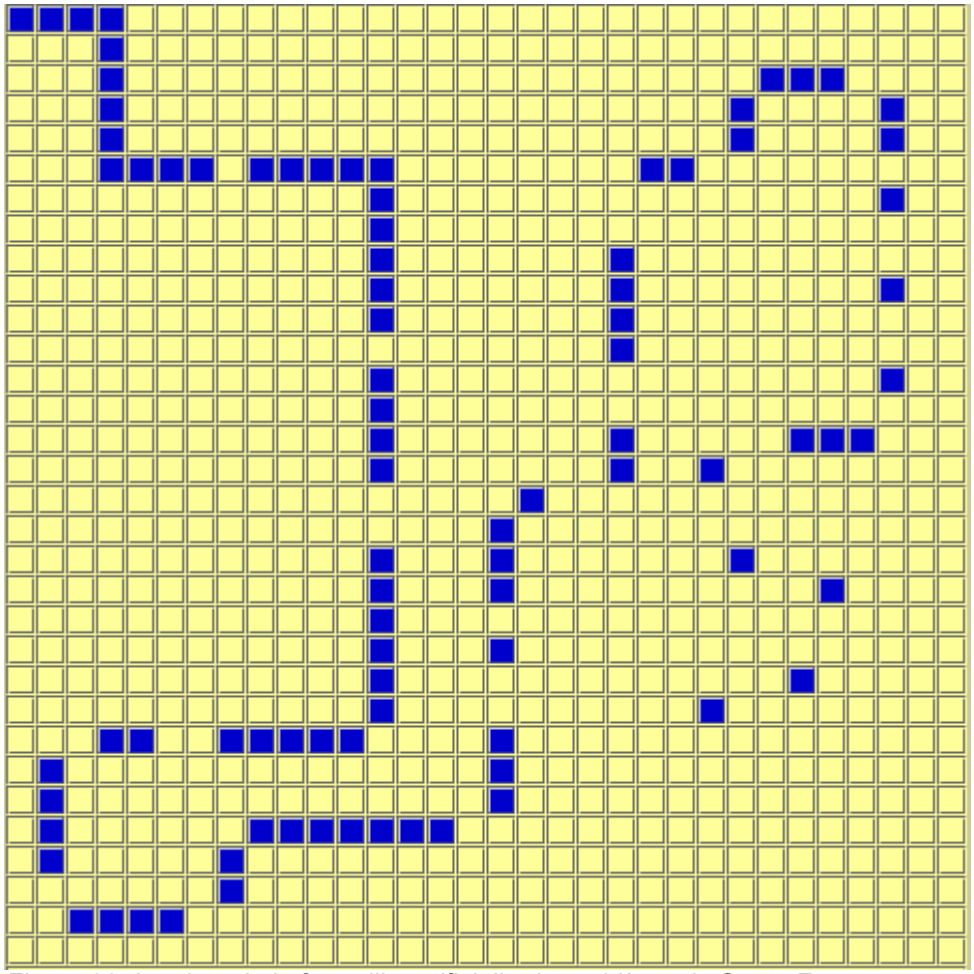

Figure 22: La piste de la fourmille artificielle du problème de Santa Fe

Ce problème est considéré comme complexe tout au moins dans le cadre de la programmation génétique [LaPo98]. Comme indiqué précédemment, dans le cadre de la résolution de ce problème, la présence des registres est inutile et nous n'aurons besoin que des instructions suivantes :

- **MOVE** : déplace la fourmi d'une case dans la direction de sa tête. Si la cellule destination contient une boulette de nourriture, cette dernière est récupérée ;
- **LEFT** : la fourmi se tourne de 45° dans le sens anti-horaire ;
- **RIGHT** : la fourmi se tourne de 45° dans le sens horaire ;
- I**F-FOOD-AHEAD** : instruction conditionnelle. La partie action de cette instruction est exécutée si une boulette de nourriture est située juste en face de la fourmi. En l'absence de nourriture, la partie action de cette instruction est ignorée ;
- **PROGN2**:permet de regrouper les deux instructions qui suivent en un seul bloc d'instructions. Cela permet typiquement de réaliser plusieurs instructions dans la partie action d'une instruction comme IF-FOOD-AHEAD ;
- **PROGN3** : même principe que l'instruction précédente à la différence que trois instructions sont regroupées au lieu de deux.

L'exécution des instructions MOVE, RIGHT et LEFT consomme une unité de temps.

Voici un exemple de programme qu'il est possible de générer pour contrôler une fourmi :

```
PROGN3{ MOVE ;
 RIGHT ;
 PROGN2{ IF-FOOD-AHEAD{ PROGN2{ IF-FOOD-AHEAD{ MOVE }
 else { IF-FOOD-AHEAD{ LEFT } else { IF-FOOD-AHEAD{ PROGN3{ LEFT ;
 IF-FOOD-AHEAD{ RIGHT } else { RIGHT } ;
 MOVE } } else { RIGHT } } } ;
 MOVE } } else { LEFT } ;
 PROGN2{ LEFT ;
 IF-FOOD-AHEAD{ MOVE } else { IF-FOOD-AHEAD{ LEFT } else { RIGHT } } } } }
```

4.4.2.2 Implantation

Les programmes pour ce problème sont des tableaux de réels. Chaque instruction est représentée par une valeur réelle unique qui est décodée de façon identique aux problèmes de régression en utilisant l'équation 5. Les différentes instructions sont décodées de manière séquentielle, la machine virtuelle qui exécute le programme de contrôle de la fourmi est modifiée de manière à permettre les sauts au-dessus d'une instruction ou d'un groupe d'instructions (cas du IF-FOOD-AHEAD).

Il est possible dans le processus de décodage de rencontrer des programmes incomplets, par exemple lorsque PROGN2 est la dernière instruction du programme ; dans ce cas les

instructions incomplètes sont simplement ignorées.

La fonction fitness du problème de la fourmi artificielle est relativement simple. Comme 89 boulettes de nourriture sont disposées sur le programme, la fitness est simplement le nombre de boulettes restantes sur le parcours. Ainsi, plus le fitness est bas, meilleur est le programme et un programme atteignant une valeur de fitness de 0 est un programme idéal.

4.4.2.3 Résultats

Les résultats sont présentés dans le tableau suivant :

	DE générationnelle			**DE stable**			**TreeDE**			**PG**		
Temps max	Fit	%hits	Eval	Fit.	% hits	Eval.	Fit.	% hits	Eval.	Fit.	% hits	Eval.
400	11,55	12,5	101008	14,65	7,5	46320	17,3	3	24450	8,87	37	126100
600	0,3	82,5	88483	1,27	70	44260	1,14	66	22530	1,17	87	63300

A la différence des expériences sur la régression symbolique, la version générationnelle obtient de meilleurs résultats par rapport à la version « stable ». La PG classique obtient les meilleurs résultats pour un maximum de 400 unités de temps, mais c'est notre version de l'évolution différentielle qui offre les meilleurs résultats lorsque la limite temporelle est fixée à 600 unités de temps.

4.5 Conclusion

Les expériences effectuées montrent que notre approche DE est compétitive face à la PG en régression symbolique, et a légèrement de meilleurs résultats que la PG lors de l'utilisation de constantes. Pour le problème de Santa Fe le GP est largement meilleur quand 400 unités de temps, mais avec 600 unités notre approche se rattrape est obtient la meilleur moyenne de fitness.

5 Conclusion

Les travaux que nous avons menés dans le cadre de cette thèse ont pour but l'amélioration de quelques techniques de la programmation génétique. Nous nous sommes penchés sur le problème qui concerne la sélection de l'ensemble des terminaux de la phase de préparation de la boucle évolutionnaire, et surtout les effets de la présence de multiples terminaux non pertinents dans cet ensemble de terminaux sur la performance de la programmation génétique. Par la suite, nous avons étudié une nouvelle approche d'amélioration de la programmation génétique en s'inspirant d'une approche différente d'optimisation qui n'est autre qu'une adaptation de l'évolution différentielle dans le cadre de programmation génétique linéaire.

Les principaux résultats des expérimentations effectuées sont présentés dans le chapitre 3, où nous avons essayé d'améliorer l'approche de [ok 2000]. Nous avons expérimenté deux formules de mesure de poids des terminaux, la formule sans cumul a obtenu de meilleurs résultats dans deux expériences et s'approche des performances du PG sans les terminaux non pertinents. Mais nous remarquons cependant une chute des performances des deux méthodes utilisées dans la troisième expérience où le nombre des terminaux pertinents est égal au nombre des terminaux non pertinents.

Le chapitre 4 présente des résultats encourageants de notre approche de programmation génétique linéaire inspirée de l'évolution différentielle, la comparaison avec la PG montre que cette approche obtient des résultats compétitifs et peut être meilleure que la PG comme dans des problèmes comme celui dit de Santa Fe.

Perspectives

Pour le problème de l'ensemble des terminaux, et les terminaux non pertinents, nous pensons qu'il serait bénéfique de tester d'autres méthodes pour le groupement des terminaux et la normalisation de leurs poids. On peut penser par exemple à la logique floue [Zad 1965], l'application d'un système d'inférence floue aux premières générations de la boucle évolutionnaire avec la valeur du poids des terminaux comme variable pourrait permettre une séparation rapide et efficace des terminaux entre terminaux pertinents et non pertinents.

En ce qui concerne l'évolution différentielle, l'adaptation de la PG avec comme moteur l'évolution différentielle n'est qu'un premier test. Nous pensons qu'il serait tout à fait possible d'améliorer la boucle évolutionnaire de DE comme l'a été celle de la PG classique.

Bibliographie

[Ang 1997] P. Angeline. Comparing Subtree Crossover with Macromutation.. Evolutionary Programming VI, 101-111, Berlin. Springer. 1997.

[Bak 1987] J. Baker. Reducing Bias and Inefficiency in the Selection Algorithm. In Proceedings of Second International Conference on Genetic algorithms and their application, pages 14-21, 1987.

[Ban 1998] W. Banzaf. Genetic programmming An. The Automatic Evolution of Computer Programs and its Applications. 240-264. 1998.

[Bar 2004] Gregory Barlow. Autonomous controller design for unmanned aerial vehicles using multi-objective genetic programming. Proceedings of the Congress on Evolutionary Computation. 2004.

[Bos 1992] B. Boser, I. Guyon, V. Vapnik. A Traning Algorithme for Optimal Margin Classifiers. COLT, pages 144-152, 1992.

[Bic 2006] Christopher Bishop. Pattern recognition and machine Learning. Springer-Verlag, 2006.

[Blu 1997.] A. Blum, P. Langley . Selection of relevant features and examples in machine learning. Artificial intelligence 97(1-2), 245–271.1997.

[Boo 1978] C. de Boor. A Practical Guide to Splines, Springer, New-York, Heidelberg, 1978.

[Bra 2007] M. Brameier, W. Banzhaf. Linear Genetic Programming. Springer, 2007

[Bre 1984] L. Breiman, J. Friedman, R. Olshen. Classification and regression trees, Belmont, CA: Wadsworth, 1984

[Bro 1999] B. Brown, S. Chen. Beta-Bernstein smoothing for regression curves with compact support, Scandinavian Journalof Statistics,26, pages 4759. 1999.

[Cur 2007] R. Curry, M. Heywood. One-Class Learning with Multi-Objective Genetic Programming. 2007.

[Cra 1985] N. L. Cramer. A Representation for the Adaptive Generation of Simple Sequential Programs. J.J. Grefenstette, editor, Proceedings of the 1st International Conference on Genetic Algorithms, pages 183-187, 1985.

[Dav 1994] S. Davies, S. Russell. Np-completeness of Searches for Smallest Possible Feature sets, Proc 1994 AAAI Fall Symposion on Relevance, AAAI Press, 37-39, 1994.

[Deb 2001] K. Deb. Multi-Objective Optimization Using Evolutionary Algorithms. Wiley, 2001.

[Dej 1975] K Dejong. An Analisis of the Behavior of a Class of Genetic Adaptive Systems. Phd Thesis, University of michigan, Ann arbor, 1975.

[Don 2006] D. Donoho. Formost large underdetermined systems of linear equations, the minimal l1-norm solution is also the sparsest solution. Comm. Pure Appl. Math., 59(7):907– 934, 2006.

[Fog 1966] L.J. Fogel, A.J. Owens, and M. J. Walash. Artificial Intelligence through Simulated Evolution. New York: John Wiley, 1966.

[Fri 1991] J. Friedman. Multivariate Adaptive Regression Splines. Annals of Statistics, vol. 19, no 1, 1991.

[Fuk 1990] K.Fukunga. Statisical patern recognition:Second edition. Morgan kofmann, 1990.

[Gor 1992] V. Gordon, D. Whitley, A. Bohn. Dataflow Parallelism in Genetic Algorithms. Proceedings of the conférence on parallel Problems Solving from Nature, 553-42. North Holland, 1992

[Gol 1989] D.E. Goldberg. Genetic Programming in Search, optimization and machine learning. Addison Wesley, 1989.

[Gru 1992] F Grueu. Genetic Synthesis of Boolean neural Network with a cell rewriting Develepmental process. In Schaffer. Proceedings of the workshop on Combinations of Genetic Algorithms and Neural Networks(COGANN92), pages 55-74. IEEE Computer Society Press.1992.

[Guy 2003] I. Guyon, A. Elisseeff. An Introduction to Variable and Feature Selection, Journal of Machine Learning Research, 3:1157-1182, 2003.

[Hol 1975] J. H. Holland. Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor, 1975.

[Iba 1995] H. Iba, H. deGaris, T; Sato. Genetic Programming with Llocal Hill-climbing. Parallel Problem Solving from Nature 3. Volume 866of Lecture Notes in Computer Science.

[Jef 1988] H. Jeffreys, B.Jeffreys. "Weierstrass's Theorem on Approximation by Polynomials" and "Extension of Weierstrass's Approximation Theory."Methods of Mathematical Physics, 3rd ed. Cambridge, England: Cambridge University Press, pages 446-448, 1988.

[Kir 1992] K. Kira, L. Rendell. The Feature Selection Problem: Traditional Methods and New Algorithm. Machine Learning: Proceeding of the eleventh International Conference. Morgan Kaufmann. 121-129, 1992.

[Koz 1992] J. Koza. Genetic Programming, on the Programming of Composants by Means of Natural Selection. MIT press, Cambridge, MA, 1992.

[Koh 1994] R.Kohavi, G.H. John K. Pfleger. Irrelevant Features and Subset selection problem. In Proc. ICML'94, pages 121-129. Morgan Kaufman, 1994.

[koh 1997] R. Kohavi , G. John. Wrappers for Feature Subset Selection. Artificial Intelligence. 97(1-2):273-324, 1997.

[Lan 1994] P langley Selection of relevant features in machine learning. Proceeding of AAAI Fall Symposium on Relevance, 1-5, 1994.

[Lan 1998] WB. Langdon, R. Poli. Why Ants Are Hard. Tech. Rep. CSRP-98-4, Universitu of Birmingham, School of Computer Science (January 98). Presented at EuroGP 1998.

[Lan 2000] W. Langdon. Size Fair and Homologous Tree Genetic Programming Crossovers. Genetic Programming and Evolvable Machines, 95-119. 2000.

[Liu, 1998.] H. Liu and H. Motoda. Feature Selection for Knowledge Discovery and Data Mining. Boston: Kluwer Academic Publishers, 1998.

[Liu 2005] H. Liu ,L. Yu. Toward Integrating Feature Selection Algorithms for Classification and Clustering, IEEE Trans. on Knowledge and Data Engineering, 17(4), pp. 491-502, 2005

[Loh 2004] J. Lohn, J. Hornby, D. Linden. Evolutionary Antenna Design for a NASA Spacecraft. Springer, Ann Arbor, 2004.

[Mcc 1959] John McCarthy. Lisp: A Programming System for Symbolic Manipulations. MIT Computation Center, 1959.

[Mit 83]. T. Mitchell, P.E. Utgoff et R.B. Banerji, "Learning by Experimentation : Acquiring and Refining Problem-solving Heuristics", Machine Learning, Palo Alto, 1983

[Mol 2002] L. Molina, L. Belanche, A. Nebot Evaluating Feature selection Algorithm, CCIA-LNCS 2504, 2002.

[Ng 2004] A. Y. Ng. Feature Selection, l1 vs. l2 Regularization, and Rotational Invariance. In ICML ’04: Twenty-first international conference on Machine learning. ACM Press, 2004.

[Nordin 1997] P. Nordin. Inductions Évolutionnaires de Programme de Code Machine Binaire et son Application. Krehl Verlag, Muenster, Allemagne. 1997.

[Ok 2000] , S Ok, K Miyashita and S Nishihara. Improving Performance of GP by Adaptive Terminal Selection. Lecture Notes in Computer Science , springerlink, pages 435-445, 2000.

[One 2003] M. O'neil, C. Ryan. Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language, Kluwer Academic Publishers, 2003.

[Pol 2008] R. Pol, W. Langdon, N McPhee. A Field guide to Genetic Programming. freely avaliable on http://www.gp-field-guide.org.uk/ 2008.

[Pol 2001] R. Polikar, L. Udpa, S. Udpa , V. Honavar. Learn++ : An Incremental Learning Algorithm for Supervised Neural Networks. IEEE Transactions on Systems, Man, and Cybernetics, vol. 31, p. 497-508, 2001.

[Qui 1983] J. R. Quinlan. Learning Efficient Classification Procedures and Their Application to Chess and Games. Machine learning : An artificial intelligence approach,1983.

[Qui 1993] J. R. Quinlan. C4.5: Programs fo Machine Learning. San Fransisco, SA: Morgan Kaufmann,1993.

[Rat 2001] A. Ratle, M. Sebag.Grammar-guided genetic programming and dimensional consistency: application to non-parametric identification in mechanics. pages 105-118, 2001. Applied soft computing.

[Rech 1973] I. Rechenberg. Evolutions strategies. Friedrich Frommann Verlag (G¨unther Holzboog KG), Stuttgart, 1973.

[Rya 1998] C. Ryan, J. J. Collins, and Michael O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. In EuroGP 1998, pages 83–96, 1998.

[Sam 1983] A. Samuel. AI, where it has been and where it is going. In IJCAI, pages 1152–1157, 1983.

[Say, 2007] Y. Sayes, I. Inza, P. Larranaga. A Review of Feature Selection Techniques in Bioinformatics, Bioinformatics, 23(19): 2507-2514, 2007.

[Sin 2002] D. Singh, M. singh, M. Markou. Feature Selection for Face Recognition Based on Data Partitioning. ICPR, 1:680-683,2002.

[Sch 1973] M..Schultz, Spline Analysis, Prentice-Hall, Series in automatic computation, 1973.

[Sch 1965] J. H. Schwefel. Kybernetische evolution als strategie der experimentellen forschung in der strömungstechnick. Master's thesis, Technical University of Berlin, 1965

[Sch 1981] J. H. Schwefel. Numerical Optimization of Computer Model. Wiley 1981.

[Sta 1986] U. Stadtmuller. Asymptotic properties of non parametric curve estimates, Periodica Mathematica Hungarica. 1986.

[Sto 1997] R. Storn , K. Price. Differential Evolution – a Simple and Efficient Heuristic for Global Optimization Over Continuous Spaces. Journal of Global Optimization 11(4), 341-359 (1997)

[Sto 1997] R. Storn , K. Price. Differential Evolution – a Simple and Efficient Heuristic for Global Optimization Over Continuous Spaces. Journal of Global Optimization 11(4), 341-359 (1997)

[Tel 1995] A Teller, M Velso. PADO: Lerneaning tree structured algorithms for orchestration into an object recognition system. Technical Report CMU-CS-95-101, Carnegie Mellon University, Pittsburgh,PA. 1995.

[Tun 1950] A. Turing. Computing Machinery and Intelligence. Mind, 59(236) :433– 460, 1950.

[Vee 2009] CB. Veenhuis. Tree Based Differential Evolution. EuroGP 2009, LNCS n°5481 pp 208-219, 2009.

[Wan 2003] Jen-Shiang Wang. Influences of Function Sets in Genetic Programming.Stanford Bookstore, 221-229, 2003.

[Zad 1965] L. Zadeh. Fuzzy Sets. Information and control, 8(3): 338_353, 1965

Index des figures