REPUBLIQUE ALGERIENNE POPULAIRE ET DOMOCRATIQUE
MINISTERE DE L'EDUCATION ET DE LA RECHRCHE SCIENTIFIQUE
UNIVERSITE ABOU BAKR BELKAID
FACULTE DES SCIENCES
DEPARTEMENT DE L'INFORMATIQUE

SUJET DU MEMOIRE DE MAGISTER:

La programmation automatique : outils de la programmation génétique et certaines de ses applications

Présenté par :

Mr. BOUGUETTAYA Ali Walid

Membres du jury:

Président	**Pr. Bereksi Reguig Fethi**	*Prof*	*Université de Tlemcen*
Examinateur	**Mr. Chikh Mohammed Amine**	*Prof*	*Université de Tlemcen*
Examinateur	**Mr. Mohammad Amine Abderrahim**	*MCB*	*Université de Tlemcen*
Encadreur	**Pr. Rahmoun Abdellatif**	*Prof*	*Université de Bel-Abbes*

ANNEE: 2011 - 2012

Remerciements

Louanges à Allah seigneur des mondes.

Et que la paix soit sur Mohammad le dernier de ses messagers.

Je remercie

Mes parents pour leurs soutient inconditionnel

Mon encadreur Monsieur RAHMOUN Abdellatif pour le temps précieux qu'il ma consacré

Les membres du jury pour m'avoir accordé leurs précieux temps durant la soutenance

Ainsi que tout ceux ou celles qui mon aidés afin d'achever ce modeste travail

Sommaire

Introduction Générale

Chapitre 01 : Concepts fondamentaux de la programmation génétique

Chapitre 02 : Concepts avancés de la programmation génétique

Chapitre 03 : Simulations et applications pratiques de la PG

Conclusion et perspectives

Références

Liste des Figures

Liste des Tableaux

Introduction

La programmation automatique sera sans nul doute le domaine de recherche le plus important en informatique dans les années à venir. Les raisons sont multiples. Alors que le matériel informatique évolue très vite, le développement de logiciels reste toujours très en retard si nous le comparons avec la demande toujours grandissante du marché. Malgré le fait que les programmeurs d'aujourd'hui disposent de moyens de production leur facilitant la tâche, comme les APIs (Application développement interface) et les environnements de développement intégré (Delphi, Visual Studio, ...), il n'en n'est pas moins que le travail du programmeur est toujours fait à la main et c'est la raison principale pour laquelle plein de projets logiciel sont obsolètes bien avant qu'ils soient livrés à leurs utilisateurs.

L'informatique et la technologie en générale ont toujours eu pour objectif de décharger les humains de se qui est fait à la main. Automatiser la programmation des logiciels pourra apporter un nouveau souffle à l'industrie des logiciels. Le domaine de l'intelligence artificielle qui traite de la génération automatique des programme est appelé programmation génétique.

La programmation génétique est une technique qui fait partie des algorithmes évolutionnaires. Son principe de base est de faire évoluer en parallèle une population de programmes exécutables, en utilisant des opérateurs génétiques et en se basant sur des critères de sélection et d'évaluation bien définis. Cette technique peut aussi être utilisée pour générer des structures qui ne sont pas du code exécutable comme les circuits électroniques par exemple. Dans la communauté de l'intelligence artificielle, il est courant d'utiliser l'appellation « programmation génétique » pour

désigner tout système d'apprentissage automatique qui fait évoluer des arbres. [Wolfgang - 1998]

Les premiers travaux concernant la génération automatique de programmes remontent aux toutes premières années de l'intelligence artificielle. Ainsi, en 1958, Le chercheur Friedberg a tenté de résoudre certain problèmes très simples en essayant de générer des programmes informatiques de manière automatique. [Robilliard] Les résultats obtenus alors étaient très limités à cause des performances réduites du matériel de l'époque, mais étaient tout de même en avance sur leur temps.

Au fil des années, plein de travaux ont était fait concernant l'induction par des méthodes évolutionnaires de programmes exécutables. Cependant ces travaux n'ont pas étaient reconnus comme nouveaux du fait qu'ils s'apparentaient à une autre technique connue de l'intelligence artificielle qui est celle des algorithmes génétiques. Ce n'est que depuis le début des années 90 que la programmation génétique à commencé à être considérée réellement comme un domaine de recherche à part entière. Cela est dû surtout aux travaux de J.Koza. En 1992, il a publié un livre intitulé « **Genetic Programming : On the Programming of Computers by Means of Natural Selection** » dans lequel il a reconnue que la programmation génétique est un nouveau domaine qui semble être prometteur.

Aujourd'hui, la programmation génétique est devenue un domaine de recherche très actif. Nous pouvons constater cela en consultant le nombre toujours grandissant de nouvelles publications traitant de ce domaine. L'une des raisons qui ont fait la gloire de cette technique est la possibilité de son utilisation et son adaptation à plusieurs domaines de recherche. En plus, la programmation génétique

ne nous oblige pas à connaitre a priori les propriétés et la structure des solutions que nous recherchons.

Depuis le début des années 90, la programmation génétiques a était utilisée dans un nombre impressionnant d'applications. Nous trouvons dans la littérature plus de 5000 utilisations de cette technique dans des domaines aussi variés comme la programmation automatique ou l'apprentissage machine. [Riccardo Poli - 2008] De plus, il est courant de trouver à l'aide de systèmes à base de programmation génétique des solutions équivalentes ou meilleures en termes de qualité que des solutions qui existaient auparavant à certains problèmes techniques difficiles entre autre certaines inventions du 20ème et début du 21è^{me} siècle. Aujourd'hui et après deux décennies de recherches intensives, la programmation génétique n'a pas livrée tous ses secrets et il nous reste beaucoup à découvrir.

Ce mémoire à pour vocation de donner une présentation générale de la programmation génétique et de ses applications. Nous traiterons les aspects les plus essentiels à la maitrise de cette technique sans rentrer dans les détails jugés trop théoriques. Ce texte peut servir comme manuel de prise en main rapide à toutes personnes abordant la programmation génétique pour la première fois. Nous avons jugé utile d'accompagne ce travail par trois mini-applications qui sont : l'induction de fonctions mathématiques, la génération automatique des circuits logiques ainsi que la résolution symbolique des équations polynomiales.

Notre travail suit le plan suivant :

Le chapitre 1 présente les concepts essentiels de programmation génétique. Nous avons traité dans ce chapitre assez de concepts pour qu'une personne ayant

bien assimilée ce dernier puisse d'ors et déjà implémenter un système génétique simple.

Le chapitre 2 quand à lui, présente quelques concepts avancés de la programmation génétique. Les sujets traités dans ce dernier tournent autour de deux thèmes qui sont : les méthodes alternatives d'implémentation et les techniques d'amélioration des performances.

Le chapitre 3 expose les 3 mini-applications qui ont étaient développées avec ce travail. Ces dernières servent à exposer quelques uns des concepts étudiés dans ce mémoire.

Nous terminerons ce travail par une conclusion et des perspectives.

I- Concepts fondamentaux de la programmation génétique

1- Introduction

L'un des buts majeur de l'intelligence artificielle est de donner à la machine la capacité de résoudre des problèmes jugés difficiles sans que la méthode de résolution ne lui soit indiquée. Cela équivaut à doter cette dernière de la capacité de se programmée d'elle-même. Une machine avec une telle capacité aura la possibilité d'évoluer toute seule en s'adaptant à son environnement. La programmation génétique est la méthode par excellence pour parvenir à ce but ambitieux.

Nous pouvons définir la programmation génétique comme étant une technique évolutionnaire qui permet de résoudre des problèmes difficiles en induisant de manière stochastique des programmes exécutables. Une fois exécutés, ces programmes pourront résoudre le problème posé.

2- L'avantage des programmes exécutables

Les méthodes traditionnelles de l'intelligence artificielle ne cherchent pas de solutions sous forme de programmes exécutables. Par exemple, les algorithmes génétiques font évoluer des chaînes numériques de largeur fixe. Les réseaux de neurones quand a eux utilisent des vecteurs de coefficients pour leurs arcs. D'autres techniques utilisent d'autres structures algorithmiques qui ont des structures et des tailles fixées. Même si cette manière de faire à la vertu de donner de bonnes performances, toujours est il que ces méthodes d'optimisation souffrent de deux inconvénients majeurs :

1. **Elles ont un champ d'application restreint** : une technique d'optimisation qui utilise une structure algorithmique statique ne peut résoudre un problème que lorsque sa solution peut être facilement exprimée avec la structure en question.
2. **Elles nous imposent de connaitre la forme de la solution** : en d'autre terme, les techniques traditionnelles ne permettent pas de résoudre un problème pour lequel nous ne connaissons pas la structure et la taille de sa solution.

Toutes ces limitations n'existent pas avec les programmes exécutables car ces derniers ont la capacité de s'adapter au type et à la taille du problème traité. De plus, un programme exécutable peut être conçu pour pouvoir résoudre plusieurs problèmes tous différents.

La flexibilité des structures exécutables est due principalement aux raisons suivantes : [KOZA - 1992]

- Traitement des problèmes d'une manière hiérarchique.
- L'exécution d'actions alternatives en fonction de certaines conditions.
- L'utilisation des boucles qui permettent de pouvoir exécuter une ou plusieurs actions plusieurs fois.
- L'utilisation de la récursivité ce qui permet de décomposer un problème difficile en plusieurs sous-problèmes plus élémentaires et plus facile à résoudre.
- L'utilisation de variables et de calculs intermédiaires ce qui donne beaucoup de flexibilité.
- L'utilisation de procédures et de fonctions ce qui permet de réutiliser du code exécutable là ou c'est nécessaire.

Pour toutes ces raisons, il est possible de résoudre un grand nombre de problèmes concernant pratiquement tout les domaines techniques en induisant des programmes exécutables. Cette souplesse des structures exécutables est la principale source de polyvalence de la programmation génétique.

3- Les problèmes que la programmation génétique sait résoudre

Comme cité précédemment, la programmation génétique est capable de résoudre un nombre impressionnant de problèmes techniques. Certains de ces problèmes peuvent bien sur êtres résolus efficacement en utilisant d'autres techniques analytiques ou évolutionnaires. Afin de bien utiliser la programmation génétique, il est important de connaitre la classe des problèmes qui sont les plus adaptées à cette technique. Voici une liste assez représentative des types de problèmes que la programmation génétique sait résoudre : [Riccardo Poli - 2008]

- **Les problèmes pour lesquels les relations entre les variables du problème traité sont inconnues ou ne sont pas bien comprises** : dans le cas contraire, il serait plus intéressant d'utiliser des méthodes analytiques car elles donnent des résultats optimaux.
- **Les problèmes pour lesquels la structure et la taille de la solution ne sont pas connues** : la programmation génétique considère que la structure et la taille de la solution sont une partie du problème. Dans le cas ou la structure et la taille de la solution sont connues, d'autres techniques donnent de meilleurs résultats que la programmation génétique. par exemple, il est reconnu que les algorithmes génétiques convergent beaucoup plus rapidement que la programmation génétique parce que leurs structures évoluées sont de taille fixe.

- **Les problèmes pour lesquels nous avons de bons simulateurs concernant les problèmes traités** : Dans certains problèmes d'ingénierie, de bons simulateurs ont étaient développés et qui peuvent analyser certains systèmes complexes comme par exemple des systèmes de contrôle ou des systèmes électronique. Le savoir contenu dans ces simulateurs ne nous permet pas généralement de résoudre des problèmes inverses. Dans ces cas là, les systèmes à base de programmation génétique peuvent profiter de ces simulateurs afin de générer des solutions aux problèmes posées.
- **Les problèmes pour lesquels nous pouvons accepter des solutions approchées** : Dans la majorité des problèmes d'ingénierie, les solutions exactes ne sont pas toujours requises. Nous pouvons dans ces cas là utiliser la programmation génétique qui donne des solutions approchées mais satisfaisantes.
- **Les problèmes pour lesquels nous pouvons nous contenter de petites améliorations dans les solutions existantes** : Pour certains problèmes techniques, ils existent déjà des solutions approchées qui sont de bonnes qualités. Ces solutions ont étaient trouvées après des années de recherches et d'améliorations. Pour ce genre de problèmes, il est rare et improbables de pouvoir trouver une solution optimale du jour au lendemain mais cependant des améliorations aussi petites quelles soient sont généralement les bien venues. La programmation génétique peut être utilisée avec cette catégorie de problèmes.

Il est important aussi, de garder à l'esprit que la programmation génétique comme toute technique évolutionnaire d'ailleurs, souffrent de deux limitations qui sont :

- **La convergence vers la solution n'est pas assurée**. Cela est dû à l'aspect heuristique de ces techniques. Dans la pratique, il est même possible que parfois, la programmation génétique et les techniques évolutionnaires trouvent des solutions assez élaborées pour des problèmes compliqués, tout en échouant à trouver des solutions à des problèmes élémentaires. Pour faire face à cette limitation, nous devons pour un problème donné faire plusieurs testes avec des paramètres à chaque différents jusqu'à ce qu'une solution acceptable finie par être trouvée. Dans la pratique, les choses se font de cette manière et il est rare de pouvoir trouver une solution du premier coup.
- **La convergence vers la solution optimale n'est pas assurée** : Les techniques évolutionnaires se contentes généralement de trouver de bonnes solutions mais pas forcement les meilleures.

4- Les types de programmes générés

La programmation génétique comme sont nom l'indique induit des programmes exécutables. Il est cependant nécessaire de préciser qu'ici le mot « programme » ne se limite pas seulement aux programmes écrits en langages impératifs comme pascal et fortran. La notion de programme dans le contexte de la programmation génétique doit être comprise dans un sens plus large en englobant toutes les catégories de langages informatiques connus comme par exemple :

- Les langages logiques (PROLOG),
- les langages fonctionnels (LISP),
- les langages orientés objet (SMALLTALK)
- et bien sur les langages impératifs (Basic, C, ...).

Les structures générées par les systèmes à base de programmation génétique peuvent représenter plusieurs choses. Voici quelques exemples :

- **Des algorithmes** : Généralement exprimés avec des langages procéduraux. Par exemple des algorithmes de parcours de graphes, des algorithmes de classement, des algorithmes d'ordonnancement, etc…
- **Des formules mathématiques** : La programmation génétique est souvent utilisée pour créer des formules mathématiques qui représentent au mieux un ensemble de données numérique. Par exemple, la programmation génétique peut être utilisée pour trouver :
  - La formule d'une forme géométrique : Cercle, ellipse, …
  - Une formule représentant un modèle physique (Loi de Kepler, loi d'Ohm, …)
- **Des stratégies** : Par exemple, dans la théorie des jeux, chaque joueur doit suivre une stratégie de jeu qui correspond à une heuristique lui indiquant dans chaque situation quel choix faire afin de maximiser son score. Ce genre de stratégies peut êtres induites en utilisant la programmation génétique.
- **Des modèles** : Un modèle est une sorte de maquette représentant un objet que nous voulons construire. Par exemple, un modèle représentant un circuit électronique, ou un autre représentant un objet mécanique. La programmation génétique est tout à fait capable d'induire des modèles représentant ce genre d'objets.
- **Et bien d'autres choses**.

5- La représentation des structures évoluées

L'une des premières décisions que nous devons prendre avant de commencer à concevoir un système utilisant la programmation génétique, est le choix d'une

structure algorithmique adéquate pour représenter les programmes que nous voulons induire. Généralement, les structures évoluées par programmation génétique sont représentées par l'une des trois manières suivantes : [Wolfgang - 1998]

- Sous forme linéaire.
- Sous forme d'arbre.
- Sous forme de graphe.

Notre choix d'une structure plutôt qu'une autre va affecter entre autres:

- L'ordre de parcours des instructions formant le programme.
- L'utilisation de la mémoire.
- Ainsi que l'utilisation des opérateurs génétique.

La majorité des implémentations de la programmation génétique utilisent les arbres. La nature hiérarchique de ces structures les rend parfaitement adaptés pour représenter un programme exécutable. De plus, même si les programmes sont généralement écrits et représentés en mémoire linéairement, ils leur arrivent de passer en interne par une représentation en arbre. Par exemple, lors de la compilation, un programme écrit en langage C sera représenté lors de la phase de l'analyse syntaxique en un arbre appelé « arbre syntaxique » qui n'est qu'une autre représentation du code source initiale.

Dans ce chapitre, nous n'allons traiter que les structures en arbre. Les deux autres structures seront traitées dans le chapitre suivant.

Un arbre est une structure algorithmique constituée de nœuds et d'arcs. Les nœuds sont reliés entre eux de manière hiérarchique par le biais d'arcs. La figure (1.1) montre un exemple d'arbre simple. Ce dernier, montre deux types de nœuds :

- **Les nœuds terminaux ou feuilles** : c'est les nœuds qui n'ont pas de descendants. Ils terminent les branches.
- **Les nœuds internes** : tous les nœuds qui ne sont pas des feuilles sont des nœuds internes. Ils peuvent avoir de un à plusieurs nœuds descendants.

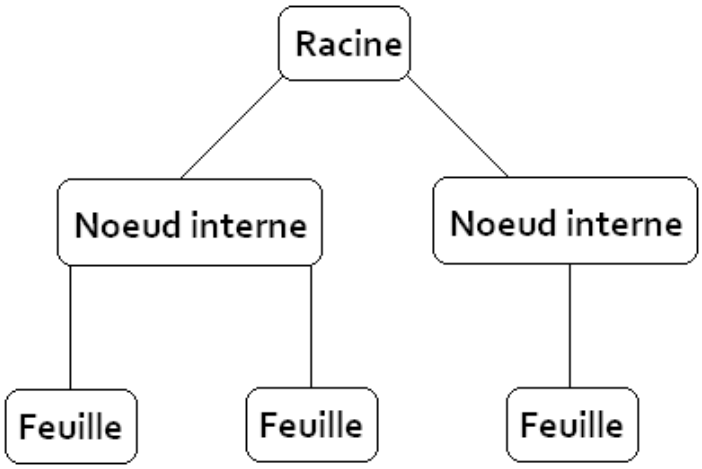


Figure 1.1 - Un arbre simple représentant les différents types de nœuds.

Le nœud le plus en haut est appelé nœud racine. C'est le seul nœud qui n'a pas de nœud parent. Chaque nœud de l'arbre est caractérisé par une profondeur. Cette dernière est calculée en comptant le nombre de nœuds à parcourir en commençant par le nœud racine pour atteindre le nœud en question qui n'est pas compté. Le nœud racine à une profondeur nulle.

Dans le cadre de la programmation génétique, il est intéressant de connaitre pour chaque arbre deux caractéristiques :

- **Le nombre de nœud** : c'est le nombre de tout les nœuds feuilles + internes.
- **La profondeur** : c'est la profondeur maximale de tous les nœuds.

Il existe des arbres pour lesquels chaque nœud a un nombre fixe de nœuds descendants. Par exemple, les arbres binaires sont des arbres pour lesquels tous les nœuds internes ont deux descendants chacun. Pour ce genre d'arbre, il est possible de calculer le nombre de nœuds à partir de la profondeur et vice versa.

En programmation génétique, il est souvent possible de trouver plusieurs manières de représenter une même structure en utilisant les arbres. Voici un exemple d'arbre simple représentant une formule mathématique.

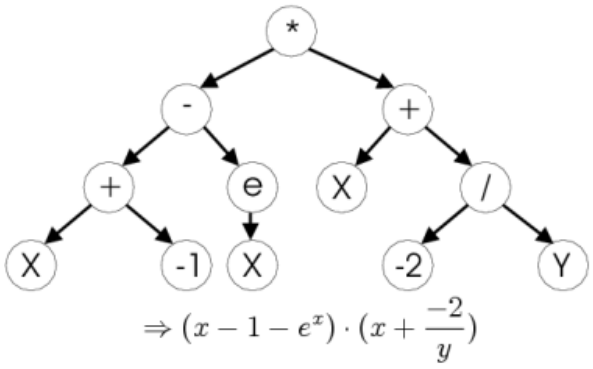


Figure 1.2 - Une formule mathématique représentée sous forme d'arbre.

Quand nous choisissons de représenter un programme sous forme d'arbre, nous devons nous mettre d'accord d'abord sur l'ordre de parcours des nœuds. Il existe trois types de parcours qui sont : [DEV-ALGO]

- **Le parcours post-fixe** : Dans cette catégorie, nous ne parcourons un nœud qu'après avoir parcouru ces descendants de gauche à droite.
- **Le parcours préfixe** : Dans cette catégorie de parcours, nous commençons par parcourir un nœud et ensuite ses descendants de gauche à droite.

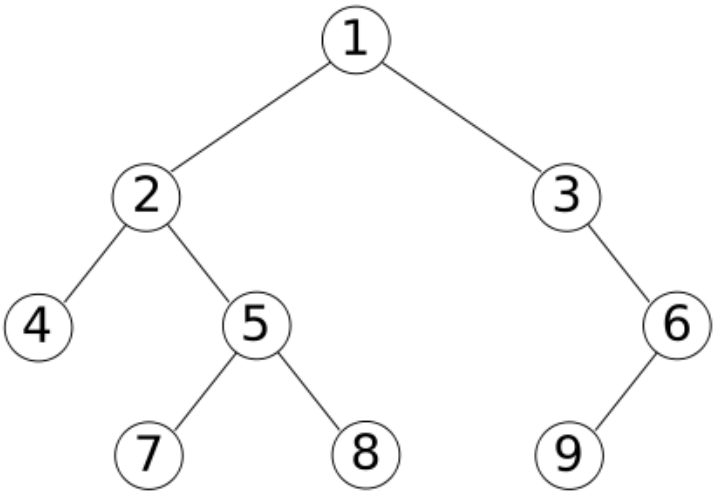
- **Le parcours infixe** : Cette catégorie ne concerne que les arbres binaires. Dans ce type de parcours, nous parcourons d'abord le descendant gauche, puis le nœud en question suivi par son descendant droite.

```
VisiterPostfixe(Arbre A) {
    Si Non_Vide(gauche(A))
       VisiterPostfixe(gauche(A))
    Si Non_Vide(droite(A))
       VisiterPostfixe(droite(A))
    Visiter(A)
}
```

```
VisiterPréfixe(Arbre A)
{
    Visiter(A)
    Si Non_Vide(gauche(A))
       VisiterPréfixe(gauche(A))
    Si Non_Vide(droite(A))
       VisiterPréfixe(droite(A))
}
```

```
VisiterInfixe(Arbre A) {
    Si Non_Vide(gauche(A))
       VisiterInfixe(gauche(A))
    Visiter(A)
    Si Non_Vide(droite(A))
       VisiterInfixe(droite(A))
}
```

Figure 1.3 - Les algorithmes des trois catégories de parcours pour un arbre binaire de racine « A » [WIKI - ARBRES]



Dans cet arbre binaire,

- Rendu du parcours infixe : 4, 2, 7, 5, 8, 1, 3, 9, 6
- Rendu du parcours postfixe : 4, 7, 8, 5, 2, 9, 6, 3, 1
- Rendu du parcours préfixe : 1, 2, 4, 5, 7, 8, 3, 6, 9

Figure 1.4 - Illustration des trois parcours pour un arbre binaire. [WIKI - ARBRES]

L'une des particularités de la représentation des programmes sous forme d'arbre est la localité de la mémoire. Pour voir cette propriété, considérons l'arbre suivant.

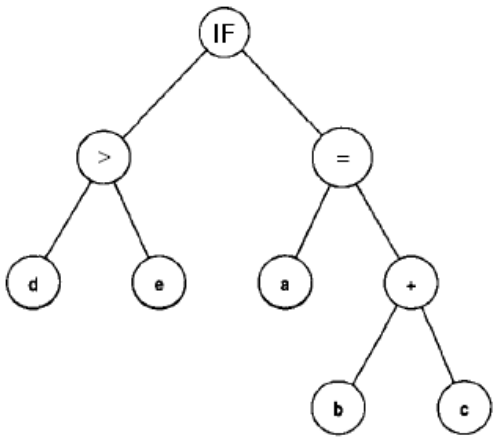


Figure 1.5 - Exemple d'arbre représentant une instruction conditionnelle.

Dans la figure (1.5), nous pouvons voir clairement que l'accès à la mémoire pour un nœud donné est limité à ses seuls descendants directs. Par exemple le nœud « > » ne peut avoir accès qu'aux deux nœuds « d » et « e ». Le nœud « + » quand à lui, ne peut accéder qu'aux deux nœuds « b » et « c ». Nous disons que dans la représentation sous forme d'arbres, la mémoire est locale. La mémoire locale est une caractéristique intrinsèque des arbres que nous ne pouvons pas changer et qu'il faut savoir gérer lors de l'implémentation.

6- **Les primitives**

En programmation génétique, les arbres formant les programmes évolués sont généralement crées en combinant de manière cohérente un nombre limités d'éléments appelés « primitives ». Ces derniers sont crées une fois pour toute au début de l'exécution. Dans le cas des arbres, il existe deux types de primitives qui sont

- **Les terminaux ou feuilles** : Ils représentent entre autres les entrées aux programmes évolués.
- **Les fonctions** : Ce sont les instructions exécutées sur les entrées.

a- Les terminaux

Dans une structure d'arbre, les terminaux sont représentés par les feuilles de l'arbre. Ces derniers peuvent êtres l'une des choses suivantes : [Wolfgang - 1998]

- Les entrées aux programmes exécutables.
- Les constantes utilisées par les programmes.
- Les fonctions sans arguments qui ont un effet de bord.

Dans la majorité des applications de la programmation génétique, les entrées servent à fournir les données d'apprentissage au système lors de la phase d'apprentissage ainsi que les données de teste lors de la phase de validation. Ainsi, à la différence des autres techniques d'apprentissage automatique, la programmation génétique a la particularité d'intégrer les données d'apprentissage dans son ensemble de primitives.

Les constantes peuvent êtres des nombres ou tout autre type d'objet manipulé par les programmes. Il existe un type particulier de constants appelés « les constantes éphémères ». Alors que les constantes normales sont spécifiées en extension, les constantes éphémères quand à elles, sont spécifiées en intension ou compréhension. L'ensemble éphémère le plus utilisé est sans doute celui des nombre réels.

Les fonctions sans arguments ne sont pas des fonctions au sens mathématique du terme mais correspondent plutôt aux procédures des langages impératifs. Ces fonctions ont des effets de bords car elles peuvent changer indirectement l'état de la mémoire du programme exécuté.

Pour illustrer les trois catégories de terminaux, nous allons donner un exemple simple. Nous disposons d'un robot que nous voulons doter de la capacité de se déplacé tout en évitant les obstacles. Pour cela, nous devons lui implémenter un algorithme qui sera exécuté de manière répétitive et infinie. Nous supposons pour simplifier que notre robot n'a qu'un seul capteur « C » lui indiquant la distance le séparant de l'obstacle en face de lui. En plus ce robot ne peut exécuter que deux actions possibles :

- **FR (Forward)** : Pour avancer d'une distance constante « D ».
- **RI30 (Right 30°)**: Pour tourner à droite de 30°.

Voici un exemple d'algorithme simple qui pourra faire l'affaire :

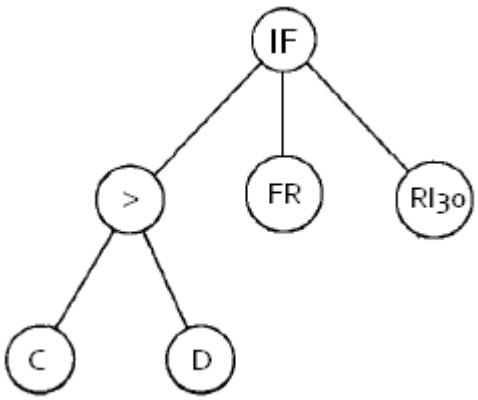


Figure 1.6 - Cette algorithme ferra que le robot avancera sans fin jusqu'à se que la distance le séparant de l'obstacle en face devienne inférieur à « C » ou il tournera à droite de 30° et recommencera l'exécution.

Dans cet exemple, l'ensemble des terminaux est le suivant : {C, D, FR, RI30}. Le terminal « C » est une entrée au programme. Sa valeur changera au fil de l'exécution de l'algorithme car le capteur renvoi à chaque itération la nouvelle distance séparant le robot de l'obstacle en face.

Le terminal « D » fait partie des constantes fournies au programme. Sa valeur ne peut pas changer car c'est une propriété intrinsèque du robot.

Les deux derniers terminaux, à savoir « FR » et « RI30 » sont des fonctions sans arguments avec effet de bord. Ce sont les commandes avec lesquelles le robot est piloté. Une fois exécutées, ces commandes vont changer les coordonnées et l'orientation du robot et ainsi changer indirectement la valeur renvoyée par le capteur C.

b- Les fonctions

Dans une structure d'arbre, les fonctions sont représentées par les nœuds internes de l'arbre. Ces fonctions s'appliquent non seulement sur les terminaux, mais aussi sur les résultats renvoyés par d'autres fonctions.

Les fonctions peuvent être de l'une des catégories suivantes : [Wolfgang - 1998]

- Les instructions
- Les opérateurs
- Les fonctions d'arité non nulle.

Les instructions peuvent être de plusieurs types mais le plus souvent ce sont des instructions de contrôle comme les instructions de choix ou de bouclage. Les opérateurs peuvent êtres des opérateurs numériques, booléens ou autres. Les fonctions quand à elles, sont souvent des fonctions mathématiques qui renvoient des valeurs numériques en fonction des entrées quelle reçoivent.

Pour illustrer les trois types de fonctions citées plus haut, nous allons donner un exemple simple. Considérons la structure exécutable suivante.

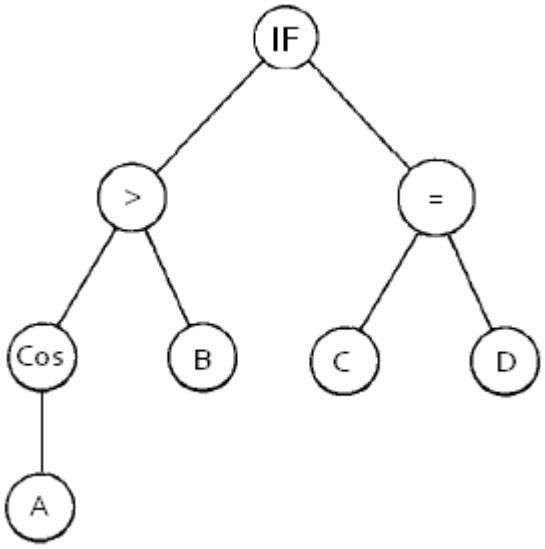


Figure 1.7 - Cet arbre représente une instruction qui affecte D à C dans certaines conditions.

L'ensemble de fonction qui figurent dans cet exemple est le suivant : {IF, >, =, Cos}. Le nœud « IF » est classé dans la catégorie « instructions ». Ce nœud représente une instruction de test et c'est lui qui déterminera le sous arbre à exécuter selon la validité de la condition.

Le nœud « > » est un opérateur. Il s'applique sur deux sous arbres qui renvoient chacun une valeur numérique. Cet opérateur renvoi deux valeurs possibles

« vrai » ou « faux ». La fonction « Cos » est une fonction au sens mathématique du terme. Elle prend et renvoi des valeurs numériques.

c- Le choix des primitives

Il faut savoir que le choix des ensembles de terminaux et de fonctions à inclure dans les primitives utilisées a un impact majeur sur les performances du système. Mal choisir les ensembles de terminaux et de fonctions ou choisir des ensembles insuffisants peut empêcher le système de converger vers la solution. D'un autre coté, choisir un ensemble trop grand peut réduire les performances du système parce que l'arbre de recherche deviendra alors trop grand.

Les primitives sélectionnées doivent avoir deux propriétés essentielles à savoir:

- La complétude (Sufficiency).
- La fermeture (Closure).

La suffisance est la propriété d'un ensemble de primitives à pouvoir représenter de manière correcte toutes les solutions possibles d'un problème donné. Par exemple, si nous voulons générer une formule mathématique, un ensemble de fonctions ne contenant que l'opérateur d'addition ne pourra pas représenter beaucoup de solutions. Pour cet exemple bien précis, il sera plus convenable d'utiliser un ensemble de fonction contenant au moins les fonctions suivantes : {+, -, *, /, Carré, Racine, Cos, Sin, Tan, Exp, Log}. Avec ces fonctions, nous pouvons induire un nombre impressionnant de formules mathématiques.

Il faut préciser aussi, que l'ensemble de fonctions et de terminaux, ne doivent pas être plus grand que nécessaire non plus. La programmation génétique est connue pour sa créativité. Elle peut si les primitives incluses sont suffisantes créer de nouvelles en combinant intelligemment ce quelle à de disponible.

La propriété de la fermeture est une propriété importante. Nous disons qu'une fonction est dotée de la propriété de fermeture quand elle peut utiliser correctement les valeurs quelle reçoit en entrée. [Riccardo Poli - 2008] L'un des exemples les plus connus d'opérateurs non doté de cette propriété est celui de l'opérateur de division. Nous savons qu'il est impossible de diviser par zéro. Cela crée un problème quand cet opérateur reçoive dans l'entrée correspondant au diviseur une valeur nulle. Dans ce cas, la structure exécutable entière devient invalide. Afin de gérer ce genre de problèmes, il est nécessaire de transformer les fonctions non fermées en les remplaçant par des fonctions équivalentes mais étendues c'est-à-dire définie pour toutes les valeurs d'entrées possibles. Dans le cas de la division par exemple, nous pouvons la remplacer par une opération appelée « division protégée » qui renvoi zéro par exemple quant il y a division par zéro.

7- **La création de la population initiale**

Comme la plupart des méthodes évolutionnaires, la programmation génétique commence son évolution par une population dite « initiale ». Cette dernière sera améliorée génération après génération en lui appliquant des opérateurs génétiques. La population initiale peut être obtenue soit :

- **Comme résultat d'une ancienne évolution**: il est rare d'obtenir la solution recherchée par la programmation génétique du premier coup. Pour cela, il est

nécessaire de doter tout système évolutionnaire d'un moyen d'enregistrer les résultats obtenus pour pouvoir les améliorer ultérieurement.

- **Aléatoirement** : C'est le plus souvent le cas.

Avant de générer une population initiale de manière aléatoire, il est nécessaire de fixer un certain nombre de paramètres qui sont :

- **Le nombre de structure maximal à inclure dans la population initiale** : Ce nombre ne doit être ni trop petit, ni trop grand. Une population trop petite risque de ne pas converger du tout vers la solution, alors qu'une population trop grande risque de prendre beaucoup trop de temps pour le faire.
- **La taille maximale des arbres inclus dans la population initiale** : Dans le cas des arbres, cette limitation se traduit soit par un nombre maximal de nœuds dans chaque arbre ou par une profondeur maximale des arbres.

Il faut savoir que l'une des caractéristiques essentielle que la population initiale doit avoir est la variation. Les arbres contenus dans cette dernières doivent être les plus variées possibles afin de permettre au système de parcourir la plus grande partie possible de l'arbre de recherche tout en évitant les minimums locaux.

Dans la littérature, il existe une multitude de méthodes d'initialisation. Dans ce chapitre nous n'allons aborder que les trois méthodes les connues qui sont : [KOZA - 1992]

- La génération d'arbres complets (Full)
- La génération d'arbres partiels (Grow)
- La méthode combinée (Ramped half-and-half)

a- La méthode des arbres complets

Cette méthode consiste à générer une population ne contenant que des arbres entiers. Un arbre entier (full tree) est un arbre d'ont touts les terminaux ont la même profondeur qui est égale à profondeur maximale de l'arbre.

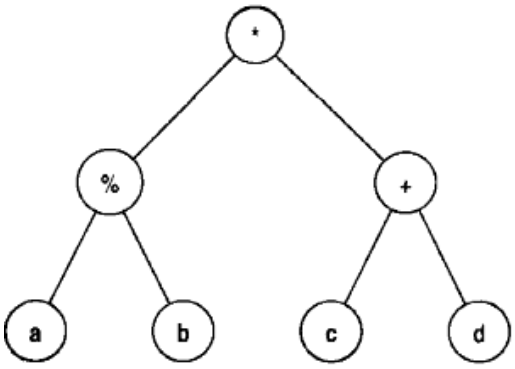


Figure 1.8 - Un exemple d'arbres entier.

b- La méthode des arbres partiels

Cette méthode consiste à générer une population contenant des arbres non entiers. Les terminaux des arbres non entiers n'ont pas tous la même profondeur. Il est à noter que les arbres non entiers apportent généralement plus de variation à la population initiale que les arbres entiers.

c- La méthode combinée

La méthode combinée appelée "Ramped half-and-half" fut inventée par Koza. Elle est une combinaison des deux méthodes précédemment évoquées. Cette dernière fonctionne en générant la moitié de la population avec la méthode des arbres entiers, et l'autre moitié avec la méthode des arbres partiels. En plus, ces générations se font par tranches de profondeurs. Par exemple si la profondeur

maximale est de 5, l'algorithme de génération créera des arbres de profondeur 1, 2, 3, 4 et 5. Cette manière de faire a la vertu de donnée une population initiale plus diversifiée que les deux méthodes précédentes.

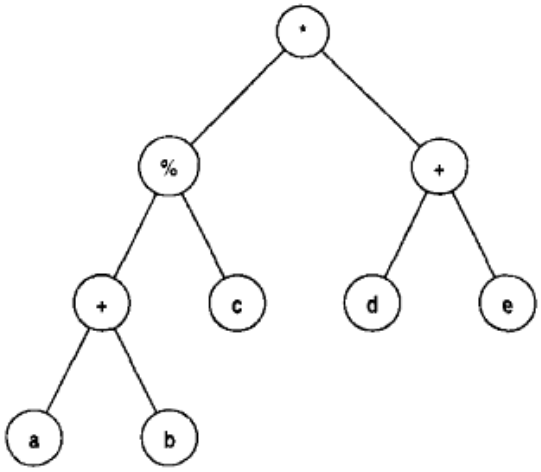


Figure 1.9 - Un exemple d'arbres partiel.

8- **Les opérateurs génétiques**

Les opérateurs génétiques sont le concept le plus important de la programmation génétique. C'est eux qui font que cette technique est plus qu'une recherche aveugle dans l'arbre de recherche. Ces derniers s'inspirent tout comme la programmation génétique elle-même du monde de la biologie. Les opérateurs génétiques les plus importants sont :

- La mutation
- Le cross-over
- La reproduction

Ces derniers peuvent fonctionner différemment selon le type de structures utilisées (linéaire, arbre ou graphe). Dans cette section, nous allons les étudier dans le cas des arbres seulement.

a- Le cross-over

Le cross-over est l'opérateur génétique le plus important. C'est lui qui est appliqué le plus souvent par rapport aux deux autres. Ce dernier est une simplification grossière du cross-over biologique. Son principe est simple. Etant donné deux individus (deux arbres parents), le cross-over va d'abord sélectionner aléatoirement deux parties (sous-arbres) de chaque individu, et ensuite les échanger. Ainsi, les deux individus obtenus (les enfants) ne sont qu'une version altérée de leurs parents.

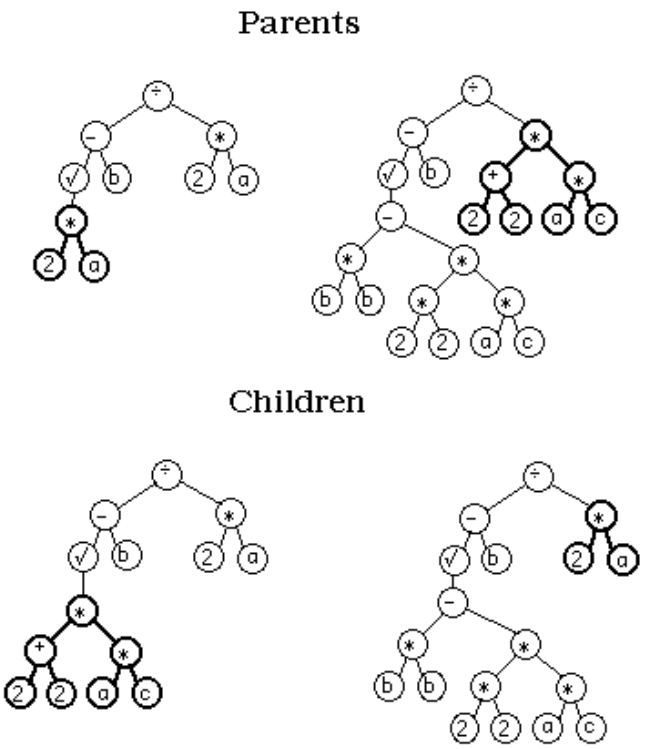


Figure 1.10 - Exemple de l'application du cross-over sur deux arbres. [GENSITE]

Il faut préciser ici que cette version du cross-over ne produit pas forcement des descendants viables. La cause est que la sélection des sous-arbres à permuter se fait aléatoirement. Ainsi, il est possible d'échanger deux parties qui ne remplissent pas la même fonction. Il existe d'autres versions du cross-over dites « informées » qui préservent la cohérence du code généré.

b- La mutation

La mutation est un autre opérateur génétique tiré du monde de la biologie. Ce dernier est plus simple à implémenter que le cross-over. Etant donné un arbre, la mutation procède d'abord par sélectionner une région aléatoire (un sous-arbre), puis de la remplacer par une autre générée aléatoirement.

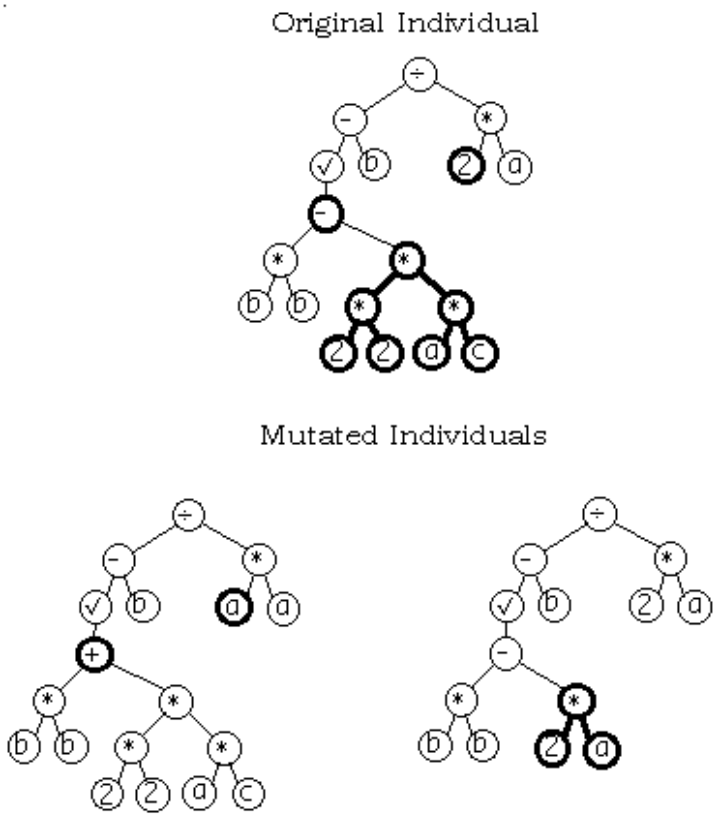


Figure 1.11 - Exemples d'applications de l'opérateur de mutation - [GENSITE]

Tout comme le cross-over, cette version de la mutation elle aussi peut générer des individus non viables. Il existe aussi pour cet opérateur, des versions informées qui préservent la cohérence du code généré.

Il faut noter que l'opérateur de mutation n'est généralement utilisé qu'à faible taux. Sa principale fonction est d'introduire plus de diversité dans la population. Cette diversité est primordiale pour échapper aux problèmes des minimums locaux.

c- La reproduction

La reproduction est un autre opérateur génétique important. Ce dernier consiste tout simplement à dupliquer un individu afin de produire un certain nombre de copies exactes. Même si généralement, cet opérateur est lui aussi utilisé à faible taux, sans la reproduction il n y aura pas d'évolution. La raison est simple. Au fil des générations, la population va accumuler de plus en plus d'individus viables qui se rapprochent de plus en plus de la solution recherchée. C'est l'opérateur de reproduction que se charge de préserver et transmettre ces bons éléments aux générations futures.

9- La fitness

La majorité des méthodes de l'intelligence artificielle se basent sur des heuristiques qui guident leurs recherches. La programmation génétique pour sa part utilise la notion de fitness. Une fitness est généralement une fonction mathématique qui nous indique le degré avec lequel un programme exécutable peut résoudre un problème donné.

C'est en utilisant la fonction de fitness, que pendant la recherche, le système génétique va décider quels sont les éléments à dupliqués, à mutés ou sur lesquels l'opérateur de cross-over sera appliqué. Pour cela mal choisir la fonction de fitness adéquate peut empêcher le système de trouver la solution recherchée ou au moins aura l'effet de réduire les performances du système.

L'une des propriétés que toute fonction de fitness doit avoir est celle de la continuité. Il faut que la fonction de fitness nous donne une idée claire et précise sur le changement de la qualité des structures exécutable que nous évoluons. Ainsi, il faut qu'une petite amélioration dans la valeur de fitness nous indique une petite amélioration dans la qualité d'un programme exécutable, ainsi qu'une grande amélioration dans la valeur de la fitness nous indique une grande amélioration dans la qualité d'un programme exécutable.

Généralement, les fonctions de fitness renvoient des nombres compris dans l'intervalle : [0, ∞ [. Il existe deux conventions possibles. Soit réserver le 0 pour le moins performant des individus, soit le réserver pour le plus performant. Dans le dernier cas, nous disons que la fonction de fitness est « standardisée ».

Il est possible parfois de limiter l'intervalle que renvoi la fonction de fitness. Dans ces cas, cet intervalle prendra la forme [0, a]. Dans le cas ou a = 1, nous disons que la fonction de fitness est « normalisée ».

Le choix de la fonction de fitness dépend principalement du problème traité. L'important, c'est de choisir une fonction qui nous indique clairement et de manière graduelle l'amélioration des individus de la population. Voici une liste non exhaustive de quelques fonctions pouvant servir de fitness pour certains problèmes :

- La quantité de nourriture mangée par une fourmi artificielle dans un jeu de vie artificielle.
- Le nombre d'exemples correctement classés dans une application de classification.
- Le nombre de d'images reconnus dans une application de reconnaissance d'images.
- Le nombre de mures évités dans une application de contrôles de robots.
- L'erreur quadratique dans une application de régression mathématique.
- Etc….

10- La sélection

En programmation génétique, la recherche passe par plusieurs générations. Au début de chaque génération, la fonction de fitness est appliquée sur toute la population. Ensuite vient l'algorithme de sélection. Ce dernier a pour objectif de décider quels sont les individus sur lesquels seront appliqués les opérateurs génétiques puis passés aux prochaines générations. L'opérateur de sélection a une importance capitale parce que c'est en choisissant les bons éléments que la recherche pourra avancer vers la solution recherchée.

Dans cette section, nous allons citer les opérateurs de sélection les plus connus.

a- La sélection basée sur la fitness (The fitness Proportional Selection)

Cet opérateur affect à chaque individu de la population une probabilité d'être sélectionné proportionnelle à sa fitness.

$$p_i = f_i / \sum_j f_j \qquad (1.1)$$

Les individus qui ont les plus grandes fitness sont ceux qui ont le plus de chance d'êtres choisis. Mais cela n'empêche pas que même les éléments les moins bons peuvent être sélectionnés parfois.

Cet opérateur fonctionne assez bien dans la majorité des cas. On lui préfère cependant un autre opérateur appelé « tournament selection ».

b- La sélection basée sur le tournoi (The Tournament Selection)

Cette méthode n'est pas basée sur la compétition entre tous les individus d'une même population, mais plutôt entre les individus d'un sous-ensemble de la population. A chaque itération, un nombre constant (appelé taille du tournoi « tournament size ») d'éléments est sélectionné. La sélection prendra effet entre ces éléments. [Wolfgang - 1998] Le meilleur individu est sélection pour lui appliqué l'opérateur génétique voulu. Pour sélectionner deux parents, deux sélections de tournoi doivent avoir lieu. [Riccardo Poli - 2008]

Cet opérateur a un certain nombre d'avantages qui sont :

- La comparaison des fitness ne se fait pas entre tous les individus de la population se qui accélère le système.
- Pour la même raison, cet opérateur est assez adapté pour le calcul parallèle.
- Puisque la sélection prend effet entre une partie des éléments de la population seulement, même les éléments ayant une fitness moyenne ont

une chance d'être choisis. Cela est nécessaire pour augmenter la diversité de la population ainsi que pour échapper aux minimums locaux.

Il est à noter que cette opérateur est l'un des opérateurs de sélection le plus utilisé dans le domaine de la programmation génétique. [Riccardo Poli - 2008]

c- La sélection basée sur le rang (The Ranking Selection)

Avec cet opérateur, la probabilité pour qu'un individu soit sélectionné est calculée en se basant non pas sur la fitness mais plutôt sur le rang de cette dernière. [Grefenstette and Baker, 1989] [Whitley, 1989] Généralement, le classement des fitness se fait soit de manière linéaire, soit de manière exponentielle.

Dans le cas linéaire, le rang est calculé avec la formule suivante : [Wolfgang - 1998]

$$p_i = \frac{1}{N}\left[p^- + (p^+ - p^-)\frac{i-1}{N-1}\right] \quad (1.2)$$

p^-/N is the probability of the worst individual being selected

p^+/N the probability of the best individual being selected

$p^- + p^+ = 2$

Dans le cas exponentielle, les probabilités sont calculées avec la formule suivante : [Wolfgang - 1998]

$$p_i = \frac{c-1}{c^{N-1}}c^N - i \quad (1.2)$$

$0 < c < 1$

d- La sélection par troncation (The Truncation Selection)

Cet opérateur commence d'abord par sélectionner un nombre (μ) de parents. Ces derniers vont engendre un nombre (λ) de descendants. A partir de ces descendants, (μ) seront sélectionnés comme parent pour la prochaine population. Cette méthode est appelée la sélection (μ, λ). Il existe une méthode similaire appelée la sélection (μ + λ) dans laquelle les parents initiaux feront partie de la génération suivante.

11- L'algorithme général

Nous allons maintenant décrire l'algorithme général, sur lequel les systèmes de programmation génétique se basent. Ce dernier va rassembler tout les éléments que nous avons vus précédemment.

Il existe deux approches utilisées par les systèmes de programmation génétique à savoir :

1. Une approche générationnelle (generational approach)
2. Une approche à flow continu (a steady-state approach)

La première approche consiste à séparer les générations les unes des autres. Dans cette approche, au début de chaque génération, une nouvelle population découlant de la précédente sera crée et traité séparément.

La deuxième approche n'utilise pas la notion de génération. Ainsi, depuis le tout début de la recherche et jusqu'à la fin, tous les individus serons considérés

comme appartenant à une même population subissant sans cesse des changements graduels.

a- Etapes préliminaires

Avant l'exécution de l'algorithme général, il est nécessaire d'abord de passer par un certain nombre d'étapes préliminaires et cela quelque soit l'approche choisie. Ces étapes sont généralement les suivantes :

- La définition des primitives : l'ensemble des terminaux + l'ensemble des fonctions.
- Le choix de la fonction de fitness appropriée.
- Le choix de l'algorithme de sélection.
- Le choix des paramètres de recherche :

    - La taille de la population initiale.
    - La taille maximale de la population.
    - La taille maximale des individus (profondeur maximal des arbres, nombre maximal de nœuds par arbre, ...).
    - Le taux de cross-over.
    - Le taux de mutation.
    - Les critères de terminaison de la recherche (seuil de fitness ou parfois le nombre maximal d'itérations).

Le choix de ces paramètres peut influer sur la vitesse de convergence de la recherche. Afin de trouver les bons paramètres il est généralement nécessaire de faire plusieurs testes avant de tomber sur la bonne combinaison. Il est nécessaire

donc de disposer d'un moyen d'enregistrer ces paramètres afin de garder les configurations les plus prometteuses.

b- L'approche générationnelle

L'approche générationnelle crée à chaque itération une nouvelle génération clairement séparée de la précédente. Voici les étapes suivies par cette approche : [Wolfgang - 1998]

1. Au début, la population est initialisée.
2. Ensuite, les programmes de la population sont évalués en utilisant la fonction de fitness. Pour chaque individu sera affecté un nombre correspondant à sa fitness.
3. Ensuite, les étapes suivantes seront répétées jusqu'à ce que la nouvelle population soit complètement remplie :

    a. Un individu ou plusieurs sont choisis de la population en utilisant l'algorithme de sélection.
    b. Les opérateurs génétiques seront appliqués sur eux avec les taux décidés dans l'étape préliminaire.
    c. Les individus résultant seront ajouté à la population de la génération suivante.

4. Si les cirières de terminaison sont remplies, nous arrêtons la recherche, sinon, nous remplaçons la génération actuelle par la prochaines et nous répétons les étapes (2-4).
5. A la fin, le meilleur individu sera présenté.

c- L'approche à flow continu

Une alternative à l'approche générationnelle est l'approche continue. Cette dernière est plus proche de se qui se fait dans la nature. En effet, dans le monde biologique, le concept de génération n'est pas très précis parce qu'il arrive souvent que dans la même population, les parents, les enfants ainsi que les petits enfants soit tous vivants.

Voici les étapes de l'approche à flow continu : [Wolfgang - 1998]

1. Au début, la population est initialisée.
2. Ensuite, un sous-ensemble de la population est sélectionné.
3. Ensuite, les programmes sont évalués.
4. Les programmes gagnants seront sélectionnés en utilisant l'opérateur de sélection.
5. Les opérateurs génétiques seront appliqués sur les programmes gagnants.
6. Les programmes perdants seront remplacés par les gagnants.
7. Les étapes (2-7) seront répétés jusqu'à satisfaction des conditions de terminaison.
8. Le meilleur individu est présenté comme résultat.

12. Conclusion

Dans ce chapitre, nous avons présenté les concepts fondamentaux de la programmation génétique. Nous avons expliqué que cette technique est utilisable dans beaucoup de domaines et peut résoudre une multitude de problèmes techniques. Nous avons montré aussi quelques méthodes simples pour implémenter un système de programmation génétique simple. Cependant, la majorité des

systèmes réels utilisant la programmation génétique, utilisent des concepts plus avancés que ceux présentés ici. Dans le chapitre suivant, nous allons passer en vue quelques uns de ces concepts avancés.

II- Concepts avancés de la programmation génétique

Dans ce chapitre, nous allons étudier certains concepts avancés de la programmation génétique sans rentrer dans les détails trop techniques.

1- Structures de représentation alternatives

En programmation génétique, les programmes évolués sont généralement représentés par des arbres. Il est cependant tout à fait possible d'utiliser deux autres structures algorithmiques pour la représentation à savoir les structures linéaires ainsi que les graphes. Nous allons dans cette section présenter ces deux structures de représentation.

a- Les structures linéaires

Dans une représentation linéaire, un programme exécutable consiste en une suite d'instructions qui se succèdent les unes après les autres. Cette manière de représenter une structure exécutable est la plus naturelle qui soit, car toutes les machines modernes ont des mémoires linéaires.

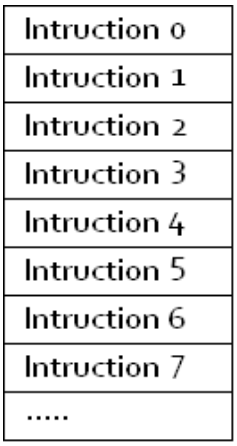


Figure 2.1 - Représentation linéaire d'un programme exécutable.

La représentation linéaire impose un ordre d'exécution simple. Les instructions sont exécutées les unes après les autres. Il est aussi possible d'introduire des instructions de saut conditionnel ou inconditionnel et dans ce cas, l'exécution des instructions ne suit plus forcement un ordre linéaire.

En générale, une structure linéaire est composée de quatre régions : [Wolfgang - 1998]

1. **L'entête (header)** : contient généralement des informations d'initialisation pour les registres.
2. **Le code exécutable proprement dit** : Est composé d'une suite d'instructions successives.
3. **Le pied de page (footer)** : contient le code nécessaire pour libérer la mémoire après la fin de l'exécution.
4. **Ainsi que le code de retour** : renvoi une valeur dans un registre spécifique.

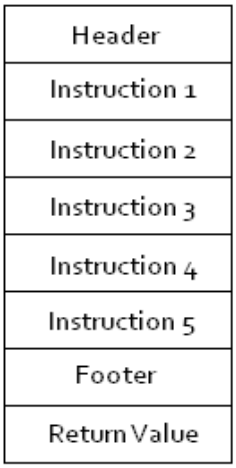


Figure 2.2 - Les différentes parties d'une structure exécutable linéaire.

L'utilisation de la représentation linéaire à deux avantages majeurs qui sont :

1. La simplicité d'implémentation.
2. L'obtention des meilleures performances.

Il existe deux types de programmations génétiques linéaires [Riccardo Poli - 2008] :

1. Programmation génétique avec code machine.
2. Programmation génétique avec code interprété.

Dans le cas de la programmation génétique avec code machine, les instructions sont directement exécutables par la machine (c.à.d. que les instructions sont en langage machine). Dans ce cas, les instructions ont le format de la machine qui les exécute. Parmi les premiers qui ont implémenté cette technique nous pouvons citer [NORDIN-1994] qui a généré du code pour des machines SUN, ainsi que [CREPEAU-1995] qui a générer du code pour les machines Z80. [Riccardo Poli - 2008] Il faut noter tout de même que même si cette manière de faire est la plus rapide en exécution, elle est la moins utilisée.

Dans le cas de la programmation génétique linéaire interprétée, les instructions sont exécutées par une machine virtuelle. Cette manière de faire est la plus utilisée grâce à sa portabilité. Dans cette catégorie, les instructions figurant dans la structure linéaire ont généralement le format de la figure (2.3).

A la différence de la représentation en arbre, les structures exécutables linéaires utilisent la mémoire de manière globale. Les données sont enregistrées

dans des registres accessibles globalement par toutes les instructions. Les instructions ont la possibilité de lire et d'écrire dans ces registres.

Output R0..R7	Arg 1 R0..R7	Opcode + − * /	Arg 2 0...127 or R0..R7

Figure 2.3 - Format des instructions d'un code exécutable linéaire. [Riccardo Poli - 2008]

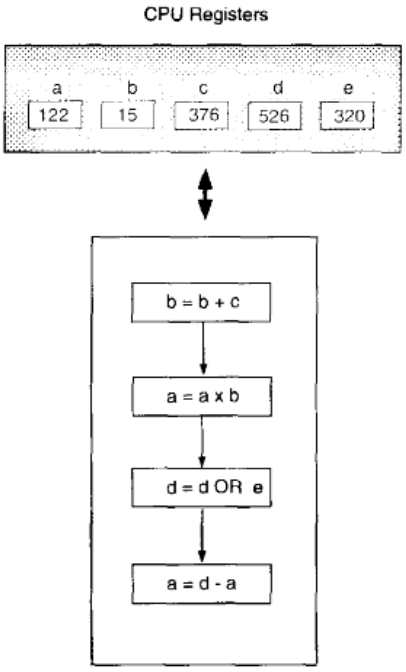


Figure 2.4 - Un exemple de programme exécutable sous forme linéaire. Les registres représentés en haut sont accessibles globalement. [Wolfgang – 1998]

Tout comme les structures en arbre, les structures linéaires ont leurs propres opérateurs de mutation et de cross-over. Il existe deux méthodes pour implémenter la mutation dans les programmes linéaires à savoir :

1. Par remplacement
2. Par altération

La méthode de remplacement consiste à sélectionner un segment aléatoire. Ce dernier sera supprimer et remplacé par un autre qui est généré aléatoirement. Les deux segments ne sont pas obligatoirement de même longueur.

La deuxième méthode consiste à sélectionner des instructions aléatoirement et de les altérer. Cette altération se fait soit en : [Wolfgang - 1998]

- Changeant aléatoirement un registre par un autre.
- Changeant aléatoirement un opérateur par un autre.
- Changeant aléatoirement une constante par une autre.

Il existe aussi un opérateur de cross-over pour les structures linéaire. Ce dernier fonctionne d'une manière très similaire à celle utilisée avec les algorithmes génétiques. Son principe de fonctionnement est le suivant :

- Deux programmes parents sont sélectionnés.
- Pour chacun de ces programmes, nous sélectionnons deux segments aléatoires.
- Ces segments sont permutés

Il existe une autre version de ce cross-over appelée le cross-over homologue de Descipulus. [FOSTER - 2001] Ce cross-over a la caractéristique de préserver la cohérence du code généré. Son principe est de ne permuter que des segments de codes se trouvant dans les mêmes régions des deux structures.

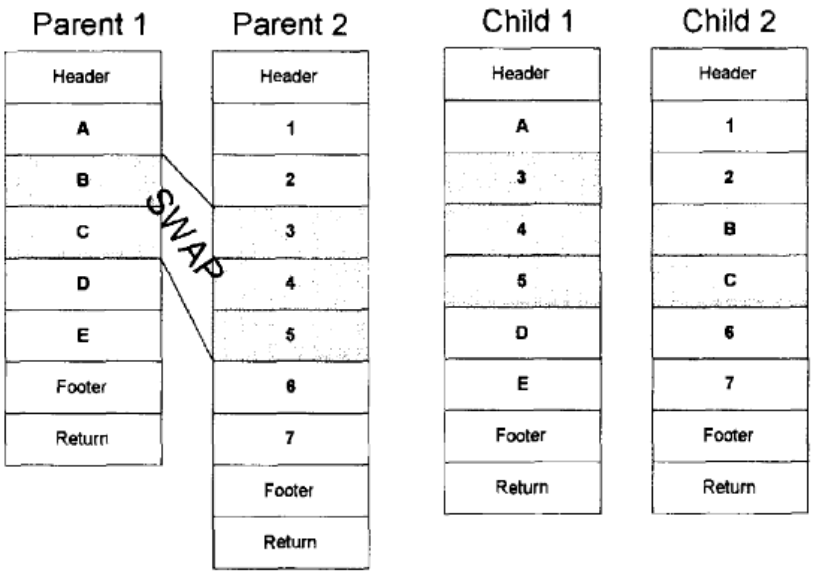


Figure 2.5 - Cross-over entre deux structures linéaires. [Wolfgang – 1998]

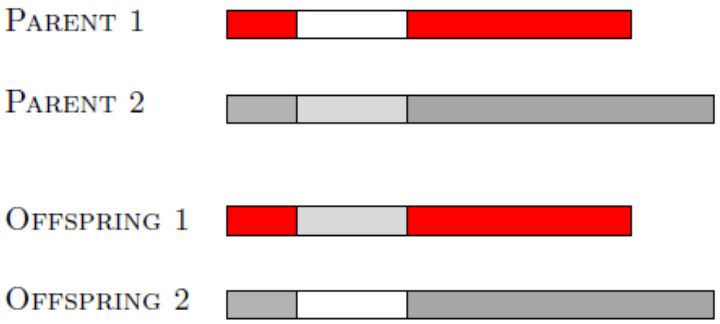


Figure 2.6 - Le cross-over homologue préserve la cohérence du code généré en ne sélectionnant que des régions parallèles de chaque programme. [Riccardo Poli - 2008]

b- La représentation en graphe

Les graphes font partis des structures algorithmiques les plus puissantes qui existent. Ils permettent de modéliser un nombre impressionnants de concepts. Leur utilisation dans le domaine de la programmation génétique a commencé depuis la moitié des années 90. [Teller and Veloso, 1995b]

A la base, un graphe est un ensemble de nœuds et d'arcs reliant ces nœuds. Quand ils sont utilisés pour représenter un programme, les nœuds représentent les instructions de ce dernier. Les arcs quand à eux, représentent le flot d'instructions ou l'ordre avec lequel les instructions seront exécutées.

La représentation des programmes sous forme de graphes a plusieurs avantages parmi lesquels nous pouvons citer :

- **Une meilleure utilisation de la mémoire** : les données ne sont pas redondantes parce que la mémoire est globale.
- **Des performances accrues** : les graphes permettent de modéliser tout les concepts des langages de programmations comme les boucles, la récursivité, l'appel de procédures, etc.... Nous pouvons booster les performances de nos programmes en implémentant ces concepts.

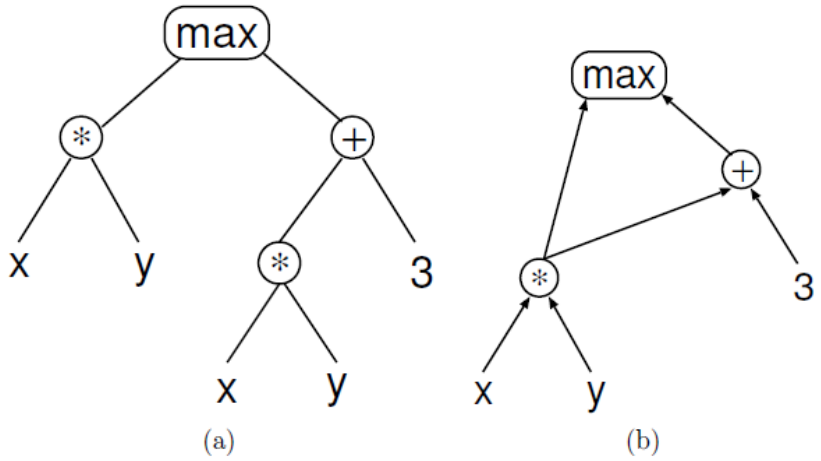


Figure 2.7 – (a)– une formule mathématique représentée sous forme d'arbre. (b) – la même formule représentée sous forme de graphe. Nous pouvons remarquer que le graphe nous permet d'économiser de la mémoire en évitant la redondance de données. [Riccardo Poli - 2008]

L'une des implémentations les plus connues des structures exécutables sous forme de graphes est PADO. PADO est acronyme de « parallel algorithm discovery

and orchestration ». Un programme PADO est un ensemble de N nœuds. Chaque nœud peut avoir autant d'arcs entrant et d'arcs sortants que nécessaire. Un nœud consiste en une partie action et une partie branchement. [Wolfgang - 1998]

Les graphes PADO utilisent la mémoire de manière globale. Ces graphes ont deux types de mémoires à savoir :

1. Une mémoire indexée.
2. Une mémoire en pile.

Comparé aux deux autres structures de représentations, les graphes imposent le moins de restrictions sur l'ordre d'exécution des instructions. Dans un graphe PADO, il ya deux nœuds spéciaux à savoir le nœud de début « START NODE» et le nœud fin « END NODE». L'exécution commence par le nœud de début et ne se termine qu'après avoir atteint le nœud de fin.

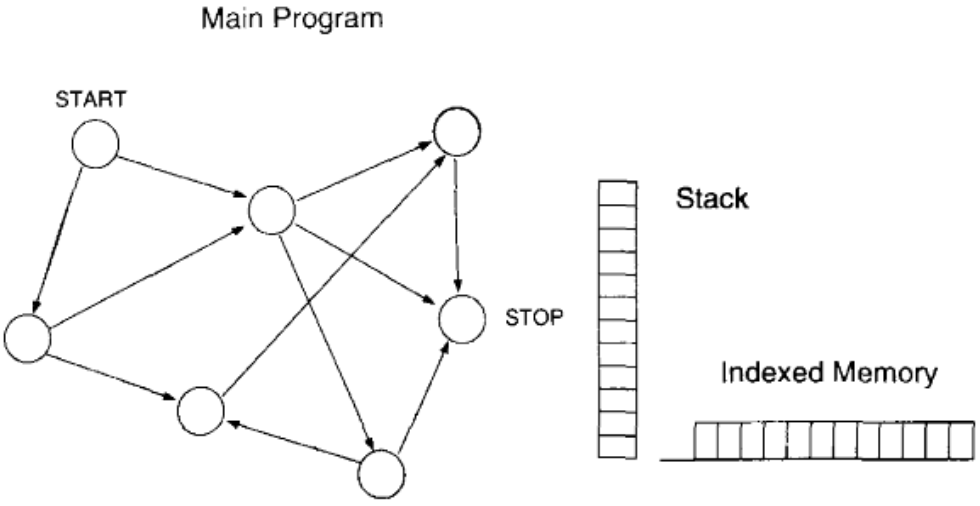


Figure 2.8 – Un graphe PADO. [Wolfgang - 1998]

2- L'initialisation par ensemencement (Seeding)

Dans le chapitre précédent, nous avons vue trois algorithmes simples pour générer aléatoirement une population initiale. Il existe dans la littérature d'autres algorithmes pour générer des populations ayant des caractéristiques bien précises. Le but, est d'influencer l'avancement de la recherche et cela dès les premières itérations. L'une de ces méthodes est la méthode d'initialisation par ensemencement (seeding)

La méthode d'ensemencement consiste à injecter dans la population initiale des structures développées à la main et qui peuvent êtres vues comme des solutions approximatives au problème traité. [R. Aler, D. Borrajo, and P. Isasi] [Holmes - 1995] [Hsu and S. M. Gustafson – 2001]. Dans ce cas là, le but de la recherche sera de trouver une amélioration possible des structures injectées.

Cette méthode à l'avantage de donner une piste de recherche dès les premières itérations. Cependant, cette technique a était critiquée car elle a deux défauts majeurs :

1. Risque de convergence prématurée.
2. Risque de perdre les structures injectées des les premières itérations.

Les structures injectées sont généralement de meilleur fitness que celle générées automatiquement. C'est la raison pour laquelle il est possible dans certains cas que dès les premières générations, la population toute entière soit complètement dominée par la même structure d'où le risque de convergence prématurée. Ce problème peut se produire dans le cas ou la population initiale est majoritairement composée de copies de la structure injectée.

Selon les paramètres de sélection choisis, il y a risque aussi que les structures injectées soient complètement broyées dès les premières générations. Cela peut être dû à l'effet parfois destructeur du cross-over. Ce cas peut se produire quand la population initiale contient très peu de copies de la solution injectée.

Pour pallier à ces deux problèmes, il faut que la population initiale soit composée de deux types de structures :

1. La moitié contenant des structures générées aléatoirement en utilisant les algorithmes déjà mentionnés : cela aura pour objectif d'assurer la diversité de la population initiale.
2. L'autre moitié doit contenir non seulement des copies exactes de la structure conçue, mais en plus des copies mutées et croisées de cette dernière.

3- Opérateurs génétiques alternatifs

Nous allons voir dans cette section d'autres versions plus sophistiqués des opérateurs de mutation et de cross-over que ceux que nous avons étudiés dans le chapitre précédent.

a- Opérateurs de mutations alternatifs

Nous avons vu dans le chapitre précédent une méthode simple pour introduire de la diversité dans la population en appliquant la mutation sur les arbres. Cette version de la mutation est appelée « mutation des sous-arbres » (subtree mutation). [Kinnear-1993] à développé une version de cet opérateur qui empêche l'arbre muté de dépasser 15% de la profondeur initial.

Il existe aussi dans la littérature, d'autres opérateurs de mutation. Voici une liste représentant quelque uns : [Riccardo Poli - 2008]

- **Mutation à taille justifiée (size-fair mutation)** : Proposé par [W. B. Langdon], cet opérateur crée des structures qui ont en moyenne la même taille que la structure initiale. La taille su code obtenu est comprise dans l'intervalle [l/2, 3l/2] ou « l » est la taille de la structure initiale.
- **Mutation à remplacement de nœuds** : Cet opérateur prend un seul nœud aléatoirement et le modifie. Pour assurer la cohérence du code obtenu, il faut que les deux nœuds possèdent le même nombre d'arguments.
- **Mutation à treuil (Hoist Mutation)** : Remplace la structure à muter en une de ces parties. Par exemple, un arbre est remplacé par l'un de ses sous-arbres. Cet opérateur diminue la taille du programme muté.
- **Mutation à réduction de taille (Shrink Mutation)**: Cet opérateur remplace un sous-arbre par un terminal aléatoire.
- **Mutation à permutation (Permutation Mutation)**: Cet opérateur sélectionne une fonction et permute ses arguments de manière aléatoire.

b- Opérateurs de cross-over alternatifs

Le cross-over est un opérateur très puissant et très important. La version du cross-over vue dans le chapitre 1 est une version simpliste. Elle consiste à permuter aléatoirement des sous-arbres. L'un des défauts majeurs de cet opérateur est le faite qu'il ne préserve pas la consistance du code généré. Il existe dans la littérature d'autres versions de cross-over qui tentent de préserver la cohérence du code généré. Parmi ces opérateurs, nous pouvons citer : [Riccardo Poli - 2008]

- **Le cross-over préservateur de context (Context-preserving crossover)** : Dans cette version du cross-over, les deux sous-arbres permutés doivent avoir les mêmes coordonnées.
- **Le cross-over à taille uniforme (Size-faire Crossover)** : Dans cette version du cross-over, un point est sélectionné aléatoirement du premier arbre. Ensuite la taille du sous-arbre sélectionné est calculée. Cette taille va guider le choix du deuxième point de cross-over en s'assurant que le deuxième sous-arbre aura cette taille. Ainsi, l'arbre obtenu aura la même taille que le premier.
- **Le cross-over intelligent (Intelligent Crossover)** : Il existe plusieurs types de cross-over dits « intelligents ». Ces derniers utilisent une approche guidée pour sélectionner des points de cross-over afin de préserver la cohérence du code obtenu. L'une de ces méthodes connues est la méthode de Zannoni. Cette dernière utilisent les techniques traditionnelles de l'intelligence artificielle afin de sélectionner les points de cross-over.

4- Contraindre la structure des arbres par les grammaires

Comme cité dans le chapitre 1, la programmation génétique permet de chercher des solutions à différents problèmes sans connaitre les structures ni les tailles de ces dernières. Il arrive cependant que la taille et la structure du programme à induire soient toutes les deux connues. Dans ce cas, il est intéressant de pouvoir forcer le système génétique à générer des solutions ayant la taille et la structure voulue. Cette manière de faire a des avantages qui sont :

- **Obtention de meilleures solutions** : En forçant la taille et la structure des structures générées à avoir celle de la solution optimale, nous avons plus de chance de tomber sur cette dernière.

- **Obtention de meilleures performances** : Puisque les structures ont des formes précises, l'arbre de recherche ne sera pas parcouru en entier. D'où les performances accrues.

Il existe plusieurs méthodes pour contraindre la structure des arbres évolués. L'une des manières les plus connues est l'utilisation des grammaires. Ces dernières sont décrites par un ensemble de règles de production.

Supposons par exemple que nous voulons induire à l'aide de la programmation génétique, des expressions arithmétiques n'utilisant que les opérations d'addition, soustraction, multiplication et division. La figure (2.9) nous montre un exemple simple de grammaires pouvant servir pour décrire ces expressions arithmétiques.

1. $Expr \rightarrow Expr\ Op$ `Number`
2. $\quad\quad\ \mid$ `Number`
3. $Op \rightarrow +$
4. $\quad\quad\ \mid -$
5. $\quad\quad\ \mid \times$
6. $\quad\quad\ \mid \div$

Figure 2.9 – Un exemple de grammaire pour des expressions arithmétiques simples.

Afin de contraindre la syntaxe des arbres évolués, la grammaire utilisée doit être utilisée en deux étapes :

1. **A l'initialisation** : La grammaire est utilisée pour créer une population de programmes respectant la syntaxe de cette dernière.
2. **Lors de l'application des opérateurs génétiques** : A chaque fois que nous voulons appliquer un opérateur génétique sur des structures évolutionnaires,

il est nécessaire de contraindre ces opérateurs afin que les structures résultantes respectent elles aussi la syntaxe de la grammaire.

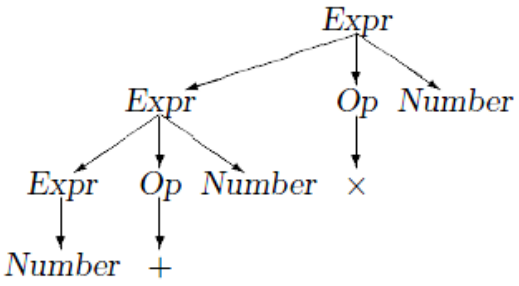


Figure 2.10 – La grammaire de la figure (2.9) peut être utilisée pour contraindre la structure des arbres induits.

5- La réutilisation du code – Les ADFs

La réutilisation du code est une notion très importante des langages de programmation. Quand ils développent de nouvelles applications, les programmeurs font généralement appel à des librairies de routines standards qui offrent différentes fonctionnalités. Afin de pouvoir implémenter la réutilisation de code, les langages de programmation utilisent les notions de procédures et de fonctions.

Les ADFs permettent de faire la même chose avec la programmation génétique à savoir réutiliser du code. ADF est acronyme de « Automatically Defined Functions » se qui signifie en français « Fonctions automatiquement définies ».

Les ADFs sont tout simplement des structures exécutables qui ont était évoluées par la programmation génétique et intégrées dans des bibliothèques afin qu'elles puissent êtres utilisées pour créés d'autres programmes ou même d'autres ADFs. Ces dernières sont utilisées comme si s'était des fonctions.

Les programmes utilisant les ADFs sont composés de plusieurs branches comme le montre la figure (2.11). Il existe deux types de branches :

1. **Les ADFs** : une ou plusieurs branches sont intégrés dans le programme afin quelles soient référencés par ce derniers.
2. **La branche RPB (result-producing branch)** : Elle représente le programme proprement dit. Elle contient une ou plusieurs références vers les ADFs se trouvant dans les autres branches.

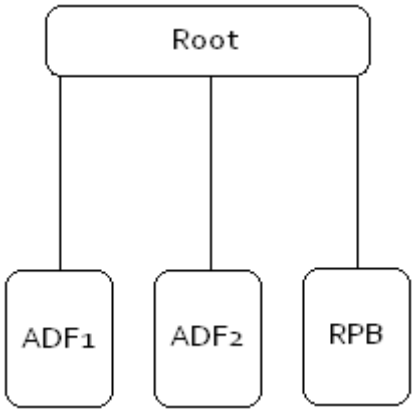


Figure 2.11 – Exemple de programme utilisant deux ADFs.

6- L'amélioration du produit du cross-over

La puissance de la programmation génétique vient en partie des opérateurs génétiques. Ces derniers, permettent aux systèmes génétiques de parcourir l'arbre de recherche de manière intelligente afin de trouver la solution recherchée. L'opérateur génétique le plus important est le cross-over. Ce dernier est celui qui est généralement exécuté avec le taux le plus élevé.

En général, le cross-over se contente de combiner aléatoirement des arbres. En faisant ainsi, il permet au système d'examiner de nouvelles combinaisons d'arbres

afin de parcourir la plus grande portion de l'arbre de recherche possible. Sans aucun doute, le cross-over a un pouvoir constructeur qui permet d'obtenir de nouveaux éléments viables. Cependant, le cross-over est aussi connu pour son pouvoir destructeur. Pour voir comment le cross-over peut parfois éloigner la recherche de la solution voulue, nous allons considérer une expérience dite de pensé (Gedanken experiement) tirée de [Wolfgang - 1998]. Nous présenterons ensuite deux solutions inspirées du monde vivant.

a- Le pouvoir destructeur du cross-over – Expérience de pensée

Considérons la figure (2.12). Cette dernière nous montre un arbre contenant 20 nœuds. Les nœuds de cet arbre sont de deux types :

- **Les nœuds sombres** : représentent le bon code. Cette partie doit être préservée car elle représente la solution voulue.
- **Les autres nœuds** : ils n'ont pas beaucoup de valeurs car ils n'appartiendront pas à la solution finale.

Si nous voulons exécuter le cross-over sur cet arbre, nous avons 19 points différents ou cet opérateur pourra opérer. Supposons au début que nous ne voulons garder que les nœuds (7-9). La probabilité que ce block est maintenu en entier est de 2/19 à savoir 10.5%. Heureusement, cette probabilité est assez petite et donc nous avons une chance de pouvoir transmettre ce block de code en entier à la génération suivante.

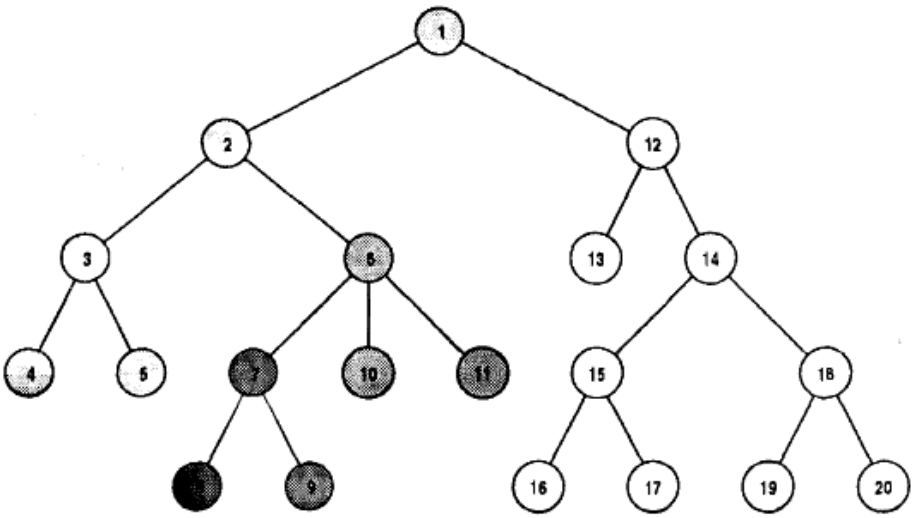


Figure 2.12 - Un exemple d'arbre avec des nœuds étiquetés. [Wolfgang - 1998]

Supposons maintenant que les nœuds (7-9) ont étaient intégré dans un block de code plus grand, lui aussi à préserver, à savoir les nœuds (6-11). Dans ce cas, la probabilité qu'un cross-over puisse détruire ce block est de 5/19 ou 26.3%. Cette probabilité est plus que le double de la dernière.

Supposons maintenant que les nœuds (6-11) sont préservés et qu'ils sont intégrés à un block à préservé encore plus grand celui des nœuds (1-11). Cette fois, la probabilité qu'un cross-over vienne détruire ce block est de 10/19 ou 52.6% ce qui n'est pas négligeable.

Supposons encore une dernière fois que ce block est quand même préservé et intégré dans la structure de la figure (2.13). Cette structure est comme nous pouvons le voir presque une solution. Il nous suffit de supprimer le nœud 12 pour obtenir une solution correct. Malheureusement, cette foi la probabilité pour qu'un cross-over vienne détruire le bon code est de 10/11 ou 90.9%.

Nous remarquons à l'issue de cette expérience de pensée, que le bon code devient de plus en plus fragile au fil de l'évolution car les structures évoluées deviennent de plus en plus grandes. Nous concluons donc que même si le cross-over a une force constructive en programmation génétique, il a aussi une force destructive. Cela vient du fait que le choix des points du cross-over est généralement aléatoire.

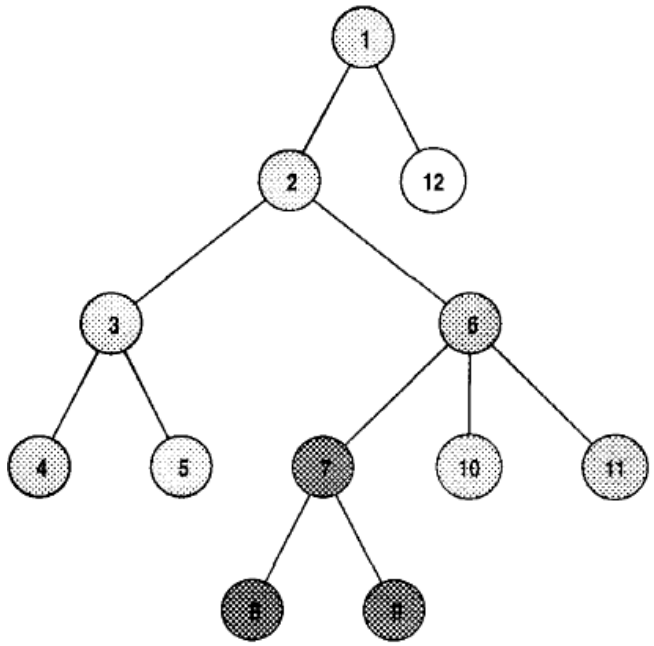


Figure 2.13 - Une structure d'arbre qui représente presque une solution. [Wolfgang - 1998]

b- Recombinaison par couvée (Brood recombination)

Dans la nature, les créatures ont tendances à produire beaucoup plus de progéniture que le nombre qui survivra. En se basant sur cette observation, Tackett a développé une méthode pour améliorer les résultats obtenus par le cross-over. [Tackett-1994] Cette méthode est assez simple. Sont principe consiste à produire plus de structures que nécessaire et ne garder que les meilleurs. Cet algorithme a un paramètre « N » appelé « taille de la couvée » ou « brood size ». Les étapes de cet algorithme sont :

- Sélectionner aléatoirement deux structures.
- Appliquer le cross-over « N » fois.
- Garder les deux meilleurs et supprimer les autres.

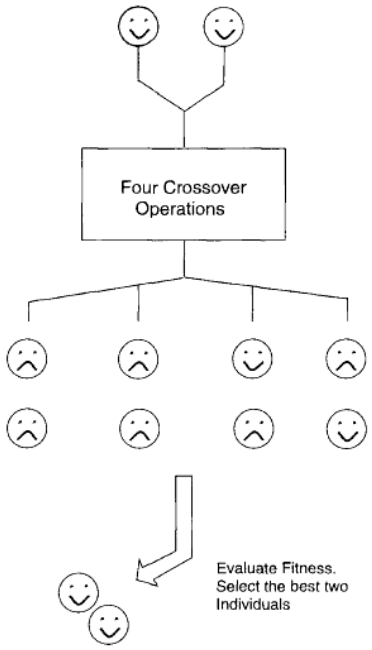


Figure 2.14 – Principe de base de la « Brood Recombination ». [Wolfgang - 1998]

Tackett a trouvé que cette méthode produit de meilleurs résultats que la méthode traditionnelle. [Tackett-1994] Cependant, cette technique a était critiquée parce qu'elle demande beaucoup de calculs. En fait, « 2N » enfants doivent êtres produits et testés au lieu de 2 pour chaque cross-over.

c- Introns définis explicitement (Explicitly Defined Introns)

Dans les cellules animales (et humaines), il semble que plus de la moitié de la longueur des chromosomes n'est pas occupées par des gènes. Ces régions sont appelées Introns. La fonction de ces dernières n'a pendant très longtemps pas était comprise. Aujourd'hui, les biologistes croient que les introns ont une fonction

importante dans la préservation de l'intégrité des gènes en gardant ensemble les bloques qui vont ensemble.

La programmation génétique elle aussi semble parfois recourir à ce genre de méthodes pour préserver les bloques de codes intéressants. Ce comportement quand il est exprimé en programmation génétique, n'est que le fruit de l'évolution.

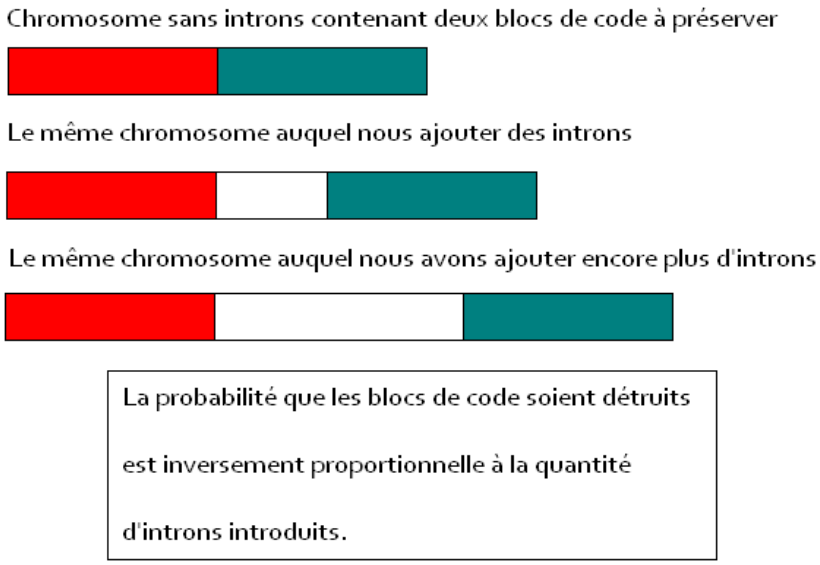


Figure 2.15 – Le rôle des introns dans la préservation de la cohérence du code.

Quand la programmation génétique utilisent les introns, elle le fait en introduisant du code inutile qui n'affect pas la fitness d'une structure.

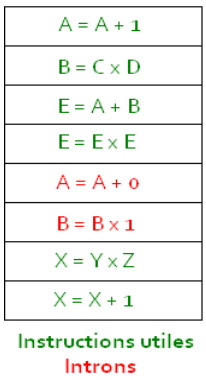


Figure 2.16 – Exemples d'introns générés par un système génétique. Ces derniers sont simplement des instructions qui n'ont aucun effet sur la fitness global du programme.

La technique des « introns explicitement définis » consiste à introduire des introns de manière explicite afin de préserver les blocs de codes. Ces introns sont représentés par des valeurs numériques qui sont introduites entre les instructions. Ces valeurs sont appelées « EDIV» ou « Explicitly defined intron value». L'opérateur de cross-over doit être modifié afin qu'il puisse prendre en considération ces valeurs numériques. Ce dernier doit tendre à sélectionner plus souvent les points de cross-over dans les régions qui ont des grandes EDIV. [Nordin et al., 1996]

Les EDIV sont ajoutées entre chaque deux nœuds (pour les arbres) ou instructions (pour les structures linéaires). Le but de cette méthode est de laisser au système génétique la possibilité de faire évoluer les vecteurs des EDIVs afin qu'il puisse identifier les blocs intéressants. Au fil des générations, le système génétique pourra adapter l'opérateur du cross-over afin qu'il préserve de plus en plus la cohérence des structures induites. Cette manière de faire correspond à faire évoluer l'évolvabilité. [Wolfgang - 1998]

7- Evaluation de la compétitivité des résultats obtenus

L'intelligence artificielle a pour objectif de concevoir des systèmes qui se comportent intelligemment comme les humains. L'un des buts majeurs de ce domaine de recherche est d'obtenir des résultats dans les domaines techniques et scientifiques qui sont comparables à ceux obtenus par des experts dans ces domaines. La programmation génétique depuis ces débuts n'à cesser de produire ce genre de solutions.

Afin de classer un résultat obtenu par la programmation génétique ou toute autre technique de l'intelligence artificielle dans cette catégorie, il est nécessaire de disposer d'un moyen pour estimer combien ce résultat est t-il compactable à ceux

obtenus par les experts. Alan Turing a proposé son célèbre teste d'imitation connu sous le nom de « le teste de Turing » ou « the Turing Test ». Le problème est que ce teste est reconnu aujourd'hui comme non utilisable. La raison est que ce genre de teste tentent d'évaluer l'intelligence qui est un concept très subjectif.

D'autres moyens d'évaluation ont étaient proposés depuis Turing. Koza à suggérer que nous devrions tourner notre attention sur le concept de compétitivité au lieu de celui de l'intelligence. [J. R. Koza, 1999] Il a proposé les critères suivants pour nous aider à décider si oui ou non, un résultat généré automatiquement peut être classé dans cette catégorie : [Riccardo Poli - 2008]

- Si le résultat était déjà découvert par les humains et breveté comme invention dans le passé.
- Si le résultat est une amélioration d'une invention brevetée.
- Si le résultat peut être considéré aujourd'hui comme étant une invention brevetable.
- Si le résultat est égale ou meilleur qu'un résultat qui était considéré come une découverte scientifique dans son temps et qui à était publié dans un magazine scientifique reconnu.
- Si le résultat peut être considéré come meilleur qu'un résultat qui était archivé dans une base de données maintenu par des experts reconnus à l'échelle internationale.
- Si le résultat peut être publié en tant que tel comme une nouvelle découverte scientifique indépendamment du fait qu'il était généré automatiquement.
- Le résultat résout un problème difficile dans son domaine.
- Le résultat est meilleur ou au moins comparable à une solution découverte par des humains pour un problème difficile pour lequel une succession de

solutions à chaque fois de plus en plus améliorées ont étaient découvertes à chaque fois par des humains.

- Le résultat a gagné une complétion reconnue qui fait intervenir des concurrents humains (soit des humains, soit des programmes créés par des humains).

Au fil des années, la programmation génétique a produit des dizaines de résultats comparables à ceux produits par des experts. Voici une liste de résultats qui ont étaient obtenus jusqu'en 2004: [Riccardo Poli - 2008]

- Création d'algorithmes quantiques incluant un algorithme meilleur que les algorithmes classiques pour les problèmes de recherche sur les bases de données et une solution pour des requêtes AND/OR.
- Création d'un programme de jeux de football compétitif pour la compétition de la RoboCup 1997.
- Création d'algorithmes pour le problème d'identification de segments transmembranaires pour les protéines.
- Création d'un réseau de classement pour sept éléments utilisant seulement 16 étapes.
- Synthèse de circuits analogiques (avec placement et routage dans certains cas), incluant :

    - Des amplificateurs à 60 et 96 décibels.
    - Des circuits pour le calcul du carré, du cube, de la racine carrée, logarithmes, la fonction gaussienne, etc….
    - Des circuits pour le control optimal d'un robot
    - Des thermomètres électroniques
    - Des circuits de conversion de courant électroniques

- La création d'une règle d'automates cellulaires pour le problème de la classification de la majorité (the majority classification problem), qui est meilleure que toutes les règles connues et obtenues par les humains.
- La synthèse d'une topologie pour contrôleurs, incluant :

    - Un crontrolleur PID (Proportional, Integrative and Derivative)
    - Un crontrolleur PID –D2 (Proportional, Integrative, Derivative and second Derivative)
    - Et bien d'autres

La majorité de ces résultats ont étaient obtenus par Koza. Depuis 2004, une compétition à lieu annuellement à la conférence internationale sur les techniques génétiques et évolutionnaire ou le prix de 10.000$ est attribuée aux projets qui produisent des résultats comparables ou meilleurs que ceux obtenus par des experts. [Riccardo Poli - 2008]

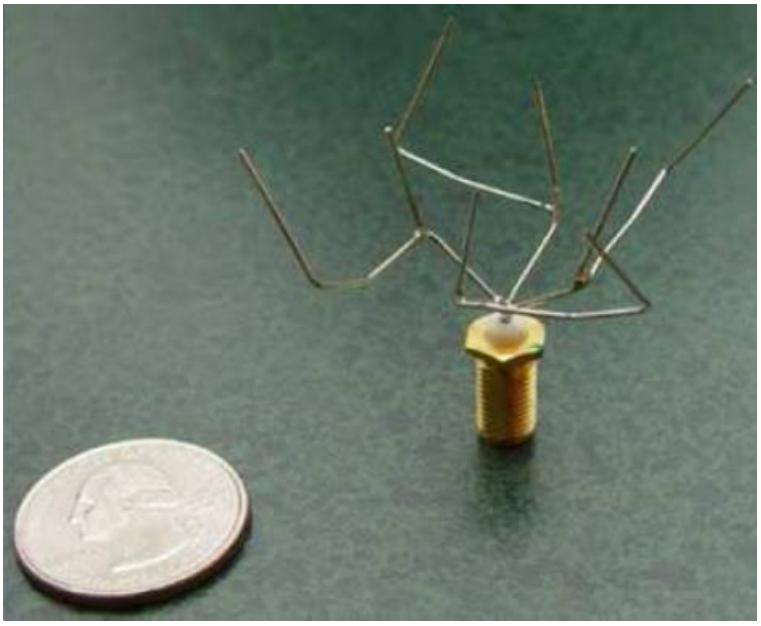


Figure 2.17 - Le prix de 2004 a était attribué à l'invention d'une antenne par programmation génétique et qui a été utilisée par la NASA Space Technology 5 Mission. [Riccardo Poli - 2008]

8- Conclusion

Dans ce chapitre, nous avons évoqués quelques uns des concepts avancés de la programmation génétique. Le but était principalement d'augmenter les performances de l'implémentation. Nous n'avons cependant pas évoqué la fusion de la programmation génétique avec d'autres techniques, comme les réseaux de neurones par exemple. Dans le chapitre suivant, nous allons continuer à étudier la programmation génétique, mais à travers trois applications pratiques que nous avons implémentés.

III- Simulations et applications pratiques de la PG

Dans ce chapitre, nous allons présenter trois mini-applications que nous avons développées pour illustrer certains concepts étudiés plus haut. Nous expliquerons pour chacune de ces applications quelques détails d'implémentations. Nous présenterons ensuite les résultats obtenues suivis de commentaires.

1- Génération automatique de fonctions mathématiques

La génération de fonctions mathématiques est l'une des applications les plus simples et les plus connues de la programmation génétique. Elle consiste à trouver une formule mathématique qui représente au mieux un ensemble de données numériques. Nous allons sur se qui suit, présenter une méthode simple pour générer des fonctions mathématique à l'aide de la programmation génétique.

a- Les primitives

Comme primitives, nous pouvons utiliser les ensembles suivants :

- **Ensemble des terminaux** : {X, R}. « R » étant la constante éphémère représentant les nombres réels.
- **Ensemble des fonctions** : {+, -, x, /, ABS, Cos, Sin, Exp, Ln}.Nous pouvons bien sur étendre cet ensemble par d'autres fonctions, mais ces fonctions peuvent êtres suffisantes dans la majorité des cas.

Pour l'ensemble des terminaux, il est possible de se limité à un singleton contenant seulement la variable « X » : {X}. Cette dernière méthode a était utilisé par KOZA [KOZA – 1992]. La raison est qu'il est tout à fait possible d'obtenir n'importe

quel nombre en combinant intelligemment la variable « x » et les quartes opérateurs arithmétiques : {+, -, x, /}.

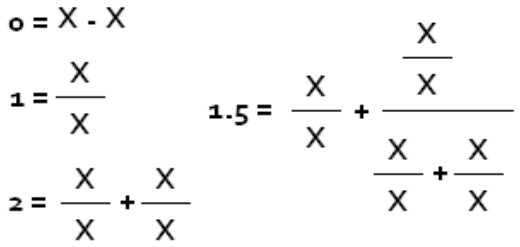


Figure 3.1 – Il est possible d'obtenir n'importe quel nombre réel en utilisant les « x » et les quatres opérateurs arithmétiques.

Cependant, l'ajout de la constante éphémère « R » peut être souhaitable. Cette dernière peut avoir deux avantages :

1. Elle peut faciliter la génération de constantes numériques et ainsi accélérer la convergence vers la solution.
2. Elle peut éviter que la taille des arbres (ou autres structures) augmente de manière incontrôlée.

L'ensemble des fonctions doit au minimum contenir les quatre opérateurs arithmétiques. Si on se limite a seulement ces derniers, la solution obtenu sera un polynôme. En ajoutant d'autres fonctions, notre système deviendra capable de modéliser des données plus compliquées.

b- La fonction de fitness

Notre objectif principal est de trouver des fonctions qui s'approchent le plus des données à modéliser. Dans ce genre d'applications, la fonction de fitness la plus

utilisée est l'erreur quadratique moyenne. Cette dernière renvoi une valeur nulle pour une fonction qui classifie les données à 100%.

$$f = \frac{1}{n}\sum_{i=1}^{n}(p_i - o_i)^2 \qquad (3.1)$$

P : Sorties renvoyées par le programme induit

O : Valeur désirée

c- Les paramètres du système génétique

Les paramètres que nous devons fixer avant de commencer la recherche de la solution sont :

- **L'intervalle des constantes réelles** : Les constantes numérique réelles ajoutées se trouvent dans l'intervalle [-R, R].
- **La taille de la population**.
- **La profondeur maximale initiale** : cette dernière ne concerne que les arbres de la population initiale.
- **La profondeur maximale des arbres**.
- **Le taux de cross-over** : Nous devons lui réserver le taux le plus élevé.
- **Le taux de mutation** : On peut se contenté généralement d'un taux ne dépassant pas 5%.
- **Le taux de duplication** : Ce dernier est obtenu en fonction des deux précédents. Duplication = 100% - (Taux Cross-over + Taux Mutation).
- **Le nombre maximal de générations**.
- **La fitness maximale acceptée** : Une fois cette valeur atteinte, le système arrêtera la recherche.

d- **Exemple**

Soit à trouver une fonction qui modélise au mieux l'ensemble de points présentés dans la figure (3.2).

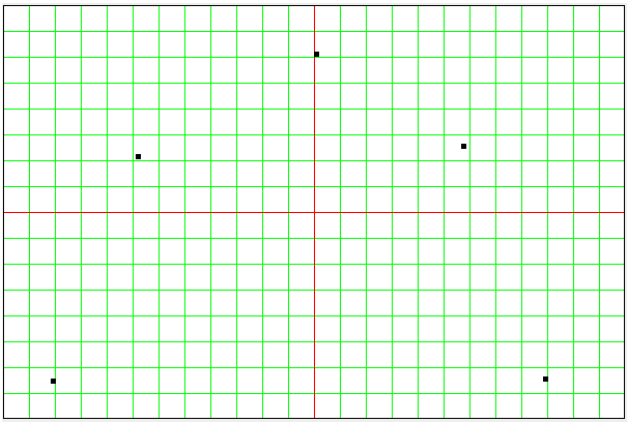


Figure 3.2 – Un ensemble de 5 points à modéliser par une fonction.

Afin d'induire la fonction recherchée, nous allons utiliser dans cet exemple les paramètres décrits dans la figure (3.3).La signification de ces paramètres est la suivante :

- **Range** : Représente la constante « R » de l'intervalle à partir duquel les constantes seront générées. Notre intervalle sera [-R, R].
- **Initial TreeDepth** : La profondeur maximale des arbres de la population initiale.
- **Maximal TreeDepth** : La profondeur maximale pour toutes les générations.
- **Number of trees** : C'est le la taille de la population.
- **Initial Num Of Full Trees** : Le nombre d'arbres entiers dans la population initiale.
- **Crossoverpercetange** : Le taux de cross-over.
- **Duplication Percentage** : Le Taux de mutation.

- **Max Num Of Evolutions** : Le nombre maximal de générations. C'est l'une des conditions d'arrête de la recherche.
- **Max Allowableerror** : L'erreur maximale tolérable. C'est aussi une condition d'arrêt de la recherche.
- **Set of Functions** :Dans cet exemple, nous avons choisi les fonctions suivantes : {+, -, x, ABS, Exp}.

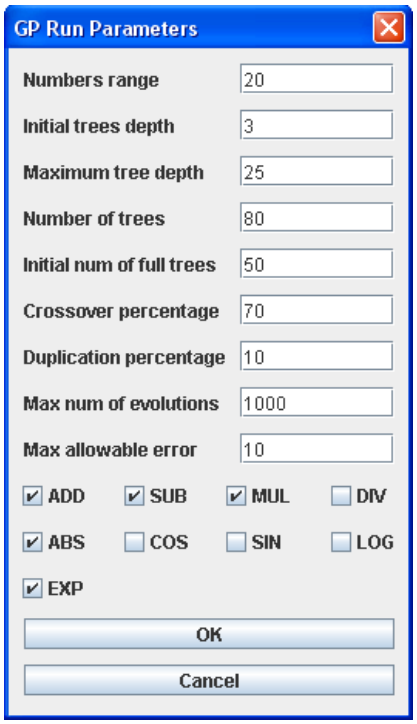


Figure 3.3 – Paramètres sélectionnés pour le système génétique.

La figure (3.4) montre le graphe de la solution obtenue après 2000 générations. La figure (3.5) montre une partie de la représentation de l'arbre généré. La figure (3.6) montre l'évolution de la fitness de la meilleure solution pendant toute la durée de la recherche.

En relançant la recherche plusieurs fois avec les mêmes paramètres, nous avons obtenu les résultats présentés dans la figure (3.7).

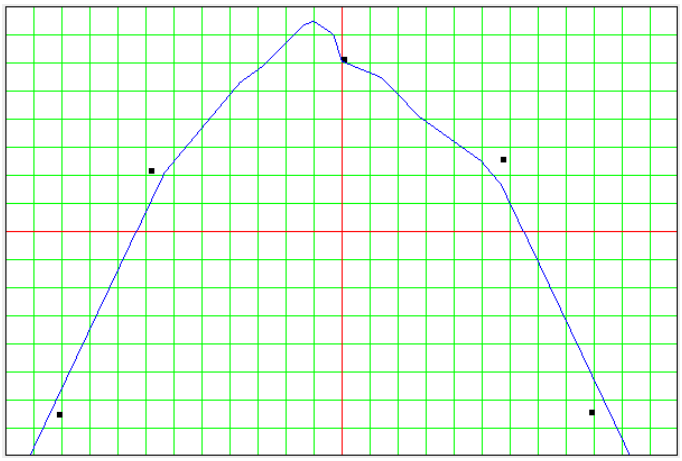


Figure 3.4 – Solution obtenue après 2000 générations.

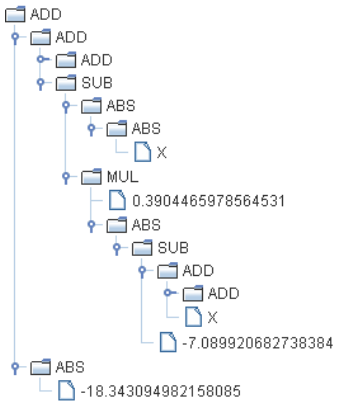


Figure 3.5 – La solution obtenue sous forme d'arbre.

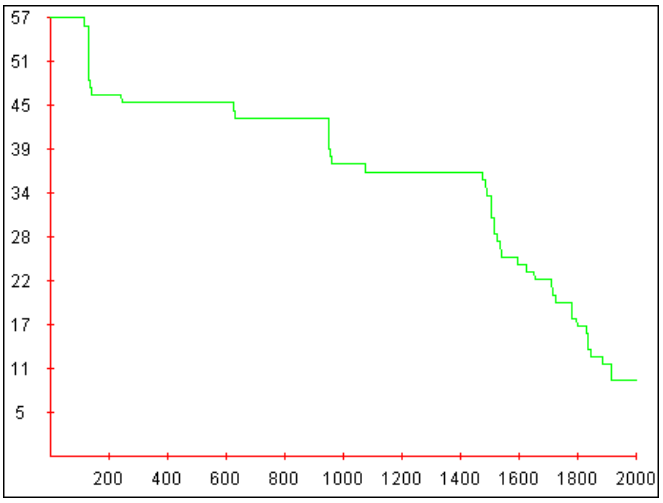


Figure 3.6 – Evolution de la fitness du meilleur élément pendant les 2000 générations.

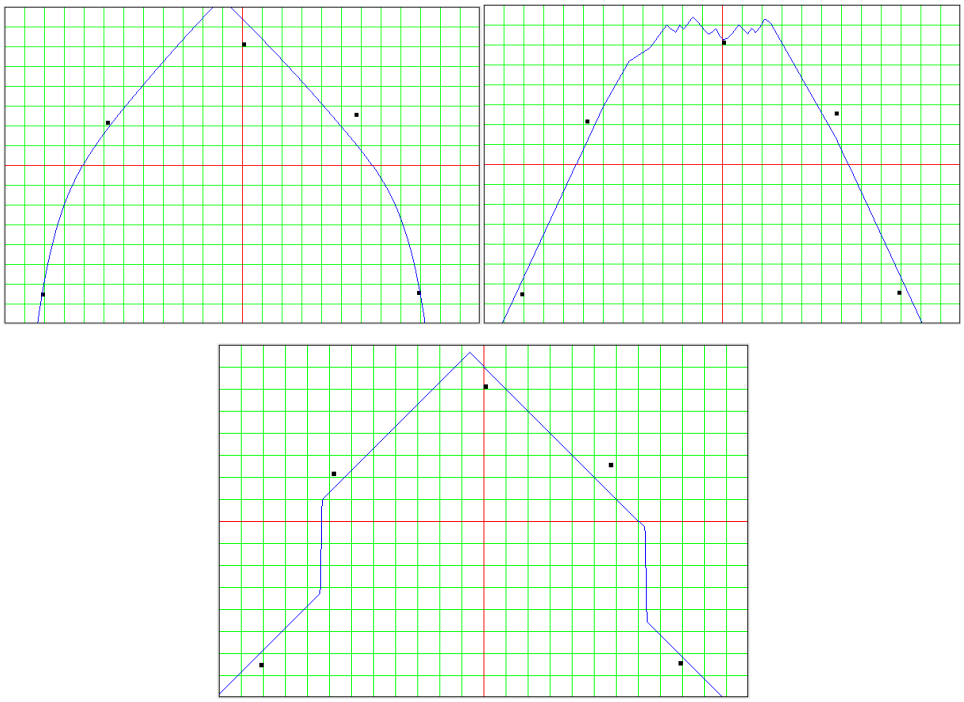


Figure 3.7 – Plusieurs autres solutions trouvées avec les mêmes paramètres.

e- **Evaluation des performances**

Il existe plusieurs méthodes pour évaluer les performances d'un système génétique. Ces dernières donnent des informations sur la quantité de calcules à faire pour obtenir la solution recherchée. La méthode que nous allons utiliser va nous indiquer la probabilité de trouver une solution en fonction du nombre de générations. Généralement, les performances sont obtenues en utilisant un grand nombre d'exemples. Nous allons pour notre part utiliser un seul exemple pour simplifier. Voici la représentation des données que nous voulons classer.

La figure (3.9) ainsi que le tableau (3.1), montrent les résultats des calculs de performances pour notre système génétique. Ces résultats nous indiquent que la majorité des recherches pour l'exemple choisi et avec les paramètres indiqués, aboutissent à une solution vers la 1000ème génération.

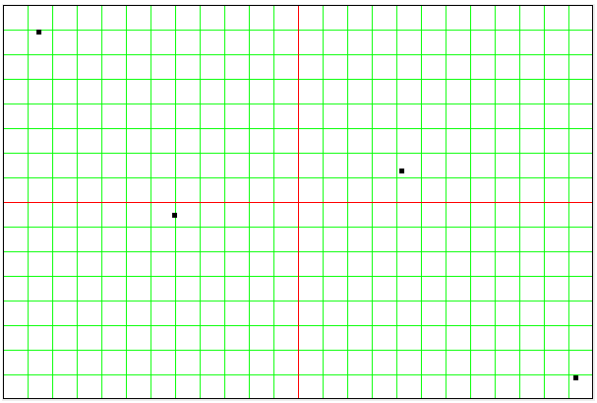


Figure 3.8 – La représentation graphique des données sur lesquelles seront calculées les performances.

Afin de connaitre l'influence des paramètres sélectionnés sur les performances du système, nous avons aussi fait quelques testes de performances en changeant à chaque fois un seul paramètre. Nous avons obtenu les graphes des figures (3.10) – (3.13).

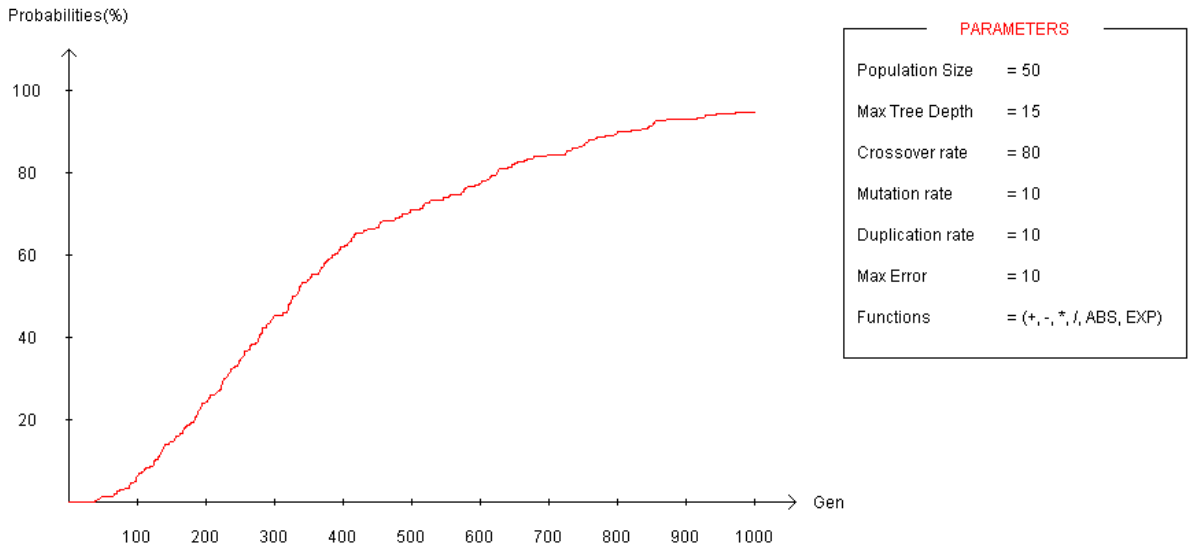


Figure 3.9 –Probabilités de classification en fonction du nombre des générations.

Générations	Taux de classification(%)
0	0
100	6.4
200	24.4
300	45.6
400	60
500	71.2
600	77.6
700	84.4
800	90
900	93.2
1000	94.8

Tableau 3.1 – Taux de classification par générations.

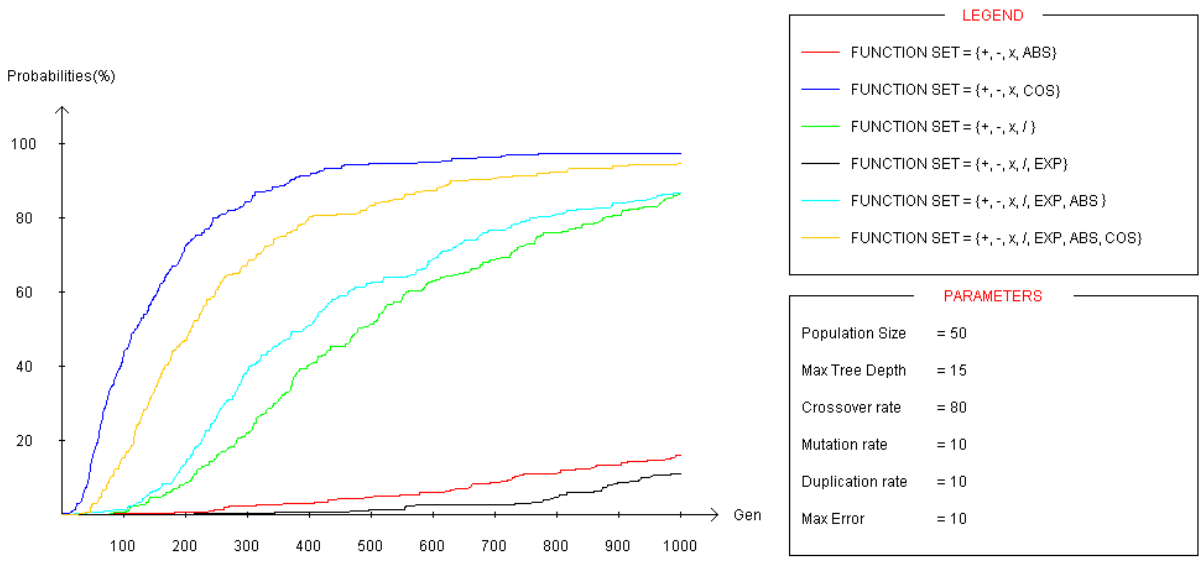


Figure 3.10 – Performances en fonction de l'ensemble de fonctions utilisés.

La figure (3.10) nous indique les performances du système en fonction de l'ensemble des fonctions sélectionné. Ce graphe nous indique que les fonctions sélectionnées ont une influence majeure sur les performances. Généralement, plus les fonctions sont nombreuses et variées, plus le système converge plus vite.

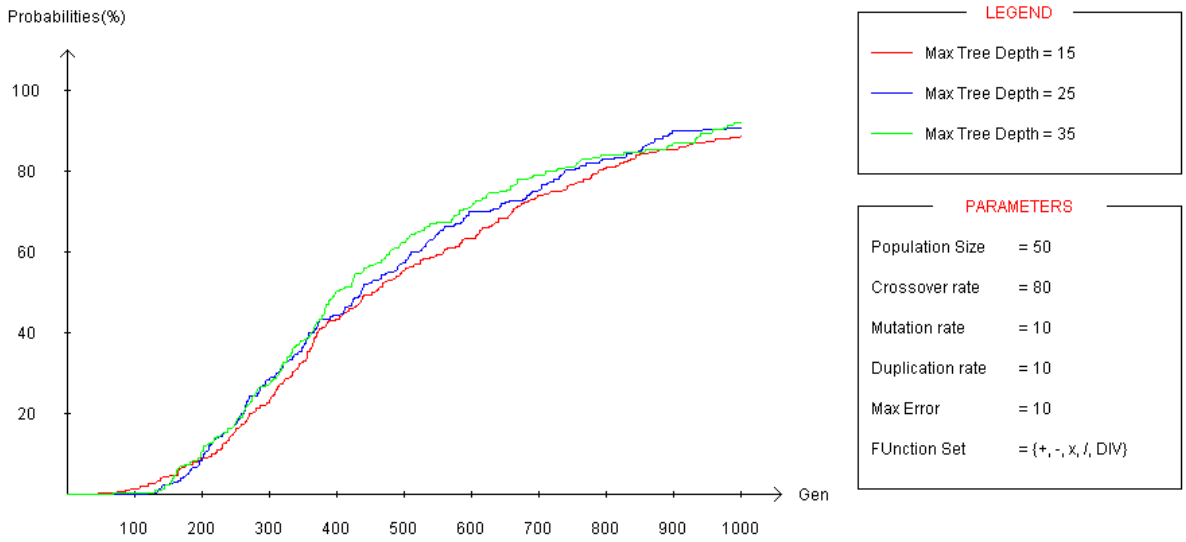


Figure 3.11 –L'influence de la profondeur maximale sur les performances du système.

Dans la figure (3.10), les meilleurs performances ont étaient obtenues par l'ensemble de fonctions contenant la fonction « COS ». Ce résultat n'est pas surprenant car en regardant les données que nous avons traitées, nous remarquons que la forme du graphe suivait une forme quelque peut ondulée. Parmi les fonctions proposées par notre système, les fonctions « Cos » et « Sin » sont les meilleurs pour modéliser ce genre de graphes.

Ce qui peut être surprenant d'un autre coté, est que l'exemple utilisant toutes les fonctions (celui représenté par la couleur orange) donne des résultats un peut moins bons que celui qui utilise la seule fonction « Cos ». Nous pouvons expliquer cela par le fait qu'à part les trois opérateurs arithmétiques (+, - et x), la fonction « Cos » est celle qui représente au mieux les données choisies par rapport aux autres fonctions. Ainsi, en ajoutant d'autres fonctions, l'arbre de recherche sera plus large car le système cherchera aussi bien les pistes utilisant seulement la fonction « Cos » que celles utilisant les autres fonctions aussi. C'est pour cette raison que le temps de convergence sera un peu plus grand.

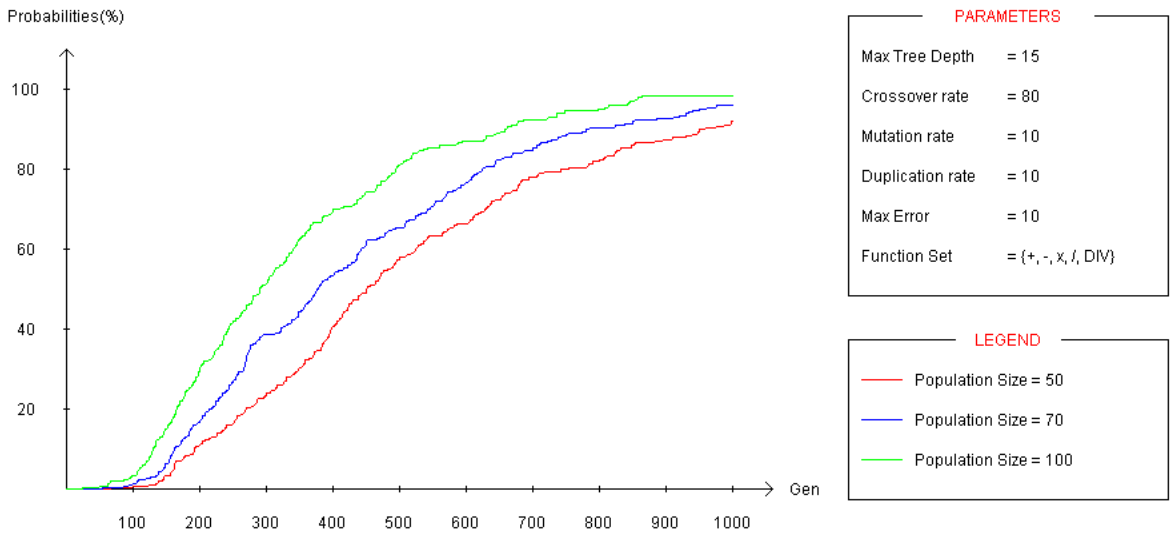


Figure 3.12 –L'influence de la taille de la population sur les performances du système.

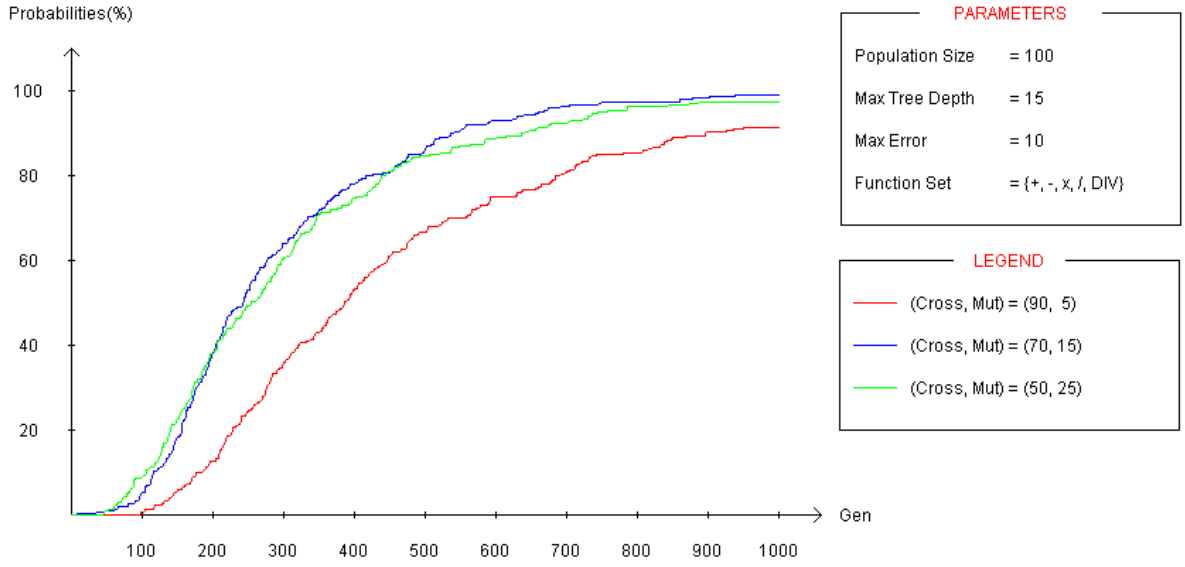


Figure 3.13 – L'influence des opérateurs génétiques sur les performances du système.

La figure (3.11) montre l'influence de la profondeur maximale des arbres sur les performances du système. Théoriquement, plus la profondeur maximale est grande, plus le système sera capable de modéliser des solutions plus élaborées. Cependant, le problème que nous avons traité est un problème simple. Cela veut dire qu'augmenter la profondeur maximale des arbres n'affectera pas beaucoup les

résultats. C'est la raison pour laquelle, les trois graphes de la figure (3.11) sont très proches les uns des autres.

La figure (3.12) montre l'influence de la taille de la population sur les performances. Normalement, plus la taille de la population augmente, plus le système aura de chance de trouver une solution. Les résultats obtenus dans la figure (3.12) confirment ce raisonnement.

La figure (3.13) montre l'influence des taux des opérateurs génétiques sur les performances du système. En programmation génétique, c'est le cross-over qui est utilisé le plus souvent. Cependant, il n'existe pas une distribution universelle pour les taux des opérateurs génétiques. Chaque problème peut avoir sa configuration optimale qui ne peut être déterminée qu'en faisant des testes.

Dans cet exemple, nous remarquons que les résultats les moins bons sont obtenus avec les paramètres [Crossover = 90%, Mutation = 5%, Duplication = 5%]. En diminuant le taux de cross-over un peut et en augmentant le taux de mutation, nous avons obtenu de meilleurs résultats. Cela indique que la population initiale n'est pas assez diversifiée et ainsi plus de mutation est nécessaire pour introduire de nouvelles structures qui pourraient éventuellement faire partie de la solution finale.

f- **Conclusion**

Il existe en analyse numérique plusieurs méthodes qui permettent d'obtenir des modèles mathématiques à partir d'ensembles de données numériques. Ces méthodes donnent généralement des bons résultats. Nous avons montré à travers notre première application, comment obtenir ce genre de modèles en utilisant la programmation génétique.

Les résultats que nous avons obtenus sont satisfaisants pour des problèmes de petites tailles. Cependant, notre approche souffre tout de même de certaines limitations qui sont :

- Les résultats obtenus à chaque lancement de la recherche sont différents.
- La taille des arbres peuvent être énorme.
- Il n y a pas de contrôle sur le type de solution obtenue. Cela veut dire que nous pouvons obtenir une équation du cinquième ordre pour des données de taille 3 par exemple. (dans ce cas, un polynôme de degré 2 est suffisent).
- Parfois, le temps de convergence est lent par rapport à la taille des données.

Il est possible d'améliorer les résultats obtenus en implémentant :

- La réutilisation de code. Nous pouvons faire cela en ajoutant un module utilise les ADFs.
- La contrainte de structure. Cela pourra forcer le système à converger plus vite et à donner des solutions ayant la forme voulue.
- La simplification des résultats. Nous obtenons cela en ajoutant un module qui simplifie la structure des arbres obtenus.

2- Génération automatique de circuits logiques

Cette application consiste à générer des circuits logiques en se basant sur leur table de vérité logique. Un circuit logique peut être vu comme une boite noire qui reçoit « n » entrés numériques binaires et qui renvoi une valeur binaire « B ». Cette dernière peut être modélisée par une fonction logique binaire de la forme

$$B = F(E_1, E_2, E_3, ..., E_n) \quad (3.2)$$

Sur ce qui suit, nous allons présenter une méthode simple pour utiliser la programmation génétique afin d'induire des circuits logiques.

a- Primitives

Les primitives que nous devons utiliser sont les suivants :

- **Ensemble des terminaux** : {E1, E2, ..., En} pour modéliser les « n » entrées du circuit logique.
- **Ensemble des fonctions** :{AND, OR, NOT, XOR, NOR, NAND}

Il est possible de se limité à un ensemble de fonctions ne contenant que les deux éléments « AND » et « NOT » ou « OR » et « NOR ».

b- La fonction de fitness

Pour le calcul de la fitness, nous utilisons le nombre de divergences entre la table logique à traiter et la table logique de la structure à évaluer. Pour calculer cette fitness, nous devons d'abord générer pour la structure à évaluer sa table logique. Ensuite, la fitness sera égale au nombre de sorties de cette table qui diffèrent de celle de la table originale. La fitness obtenue sera comprise entre « 0 » (pour une classification totale) et 2^n (pour une structure qui ne classe aucune entrée correctement).

c- Exemple

Nous allons dans cet exemple induire un circuit logique en se basant sur la table logique de la figure (3.14). Les paramètres utilisés sont ceux présentés dans la figure (3.15). Après 78 itérations, une solution avec fitness = 0 a était trouvée. Les figures (3.16) et (3.17), montrent deux représentations (arbre et circuit) de cette solution induite.

E0	E1	E2	E3	Fn
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Figure 3.14 – Table logique avec 4 entrées.

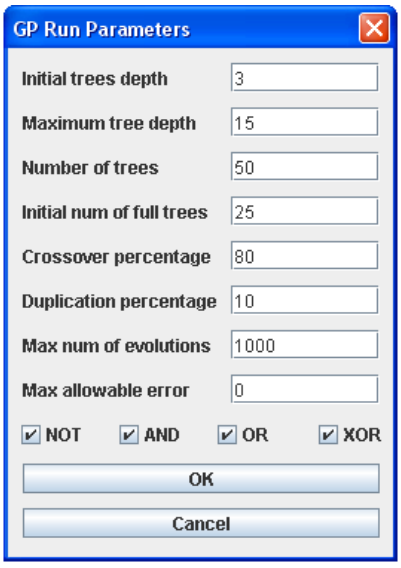


Figure 3.15 – Les paramètres utilisés pour générer le circuit logique.

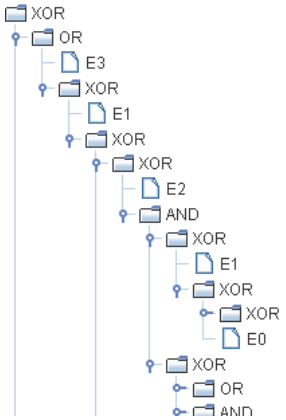


Figure 3.16 – Représentation sous forme d'arbre de la solution obtenue.

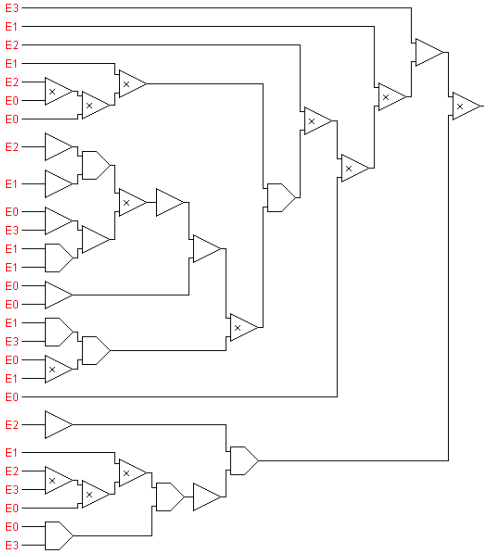


Figure 3.17 – Représentation sous forme de circuit de la solution obtenue.

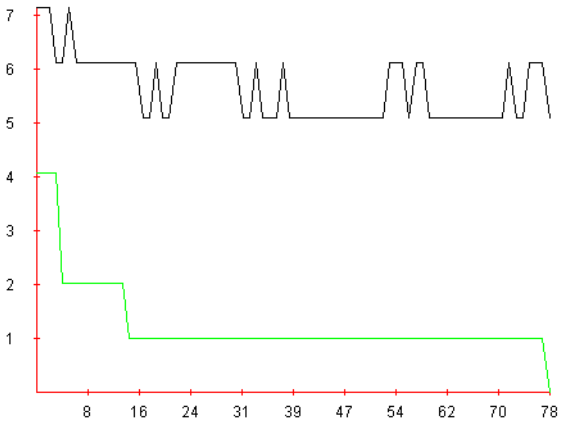


Figure 3.18 – Noir : Evolution de la fitness moyenne de toute la population. Vert : Evolution de la fitness de la meilleure structure.

d- Etude de performances

La figure (3.19) présente les probabilités de classification de la table de l'exemple précédent en fonction du nombre de générations.

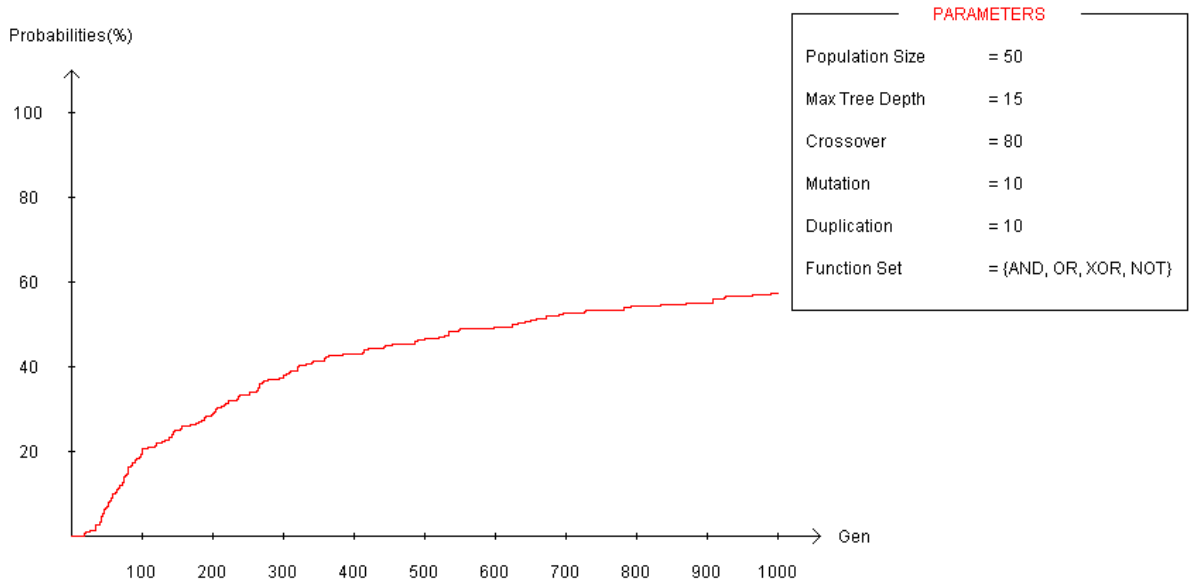


Figure 3.19 – Probabilités de classification en fonction du nombre des générations.

Générations	Taux de classification(%)
0	0
100	19.6
200	28.8
300	37.6
400	43.2
500	46.4
600	49.6
700	52.8
800	54.4
900	55.2
1000	57.6

Tableau 3.2 – Taux de classification par génération.

Afin de déterminer l'influence des différents paramètres sur les performances du système, nous avons fait un certain nombre de testes en changeant à chaque fois un seul paramètre. Les résultats obtenus sont présentés dans les figures (3.20) – (3.23).

La figure (3.20) montre l'influence de la taille de la population sur les performances du système. Cette figure nous indique qu'il n'existe pas une grande différence entre les trois tailles de population (50, 75, 100). Cependant, nous pouvons tout de même remarquer que la taille 50 donne des résultats légèrement moins bons que les deux autres.

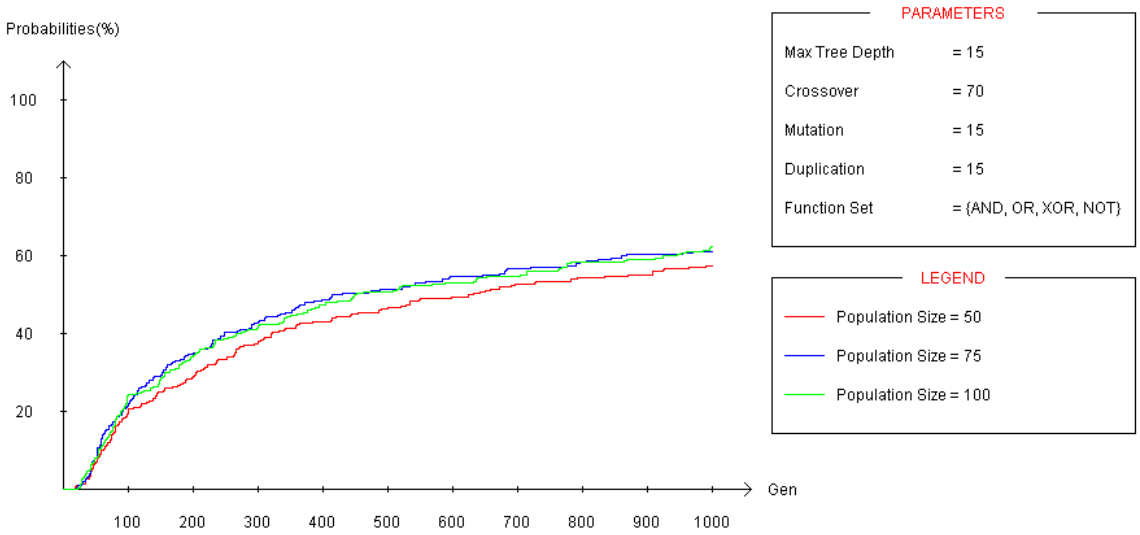


Figure 3.20 – Influence de la taille de la population sur les performances du système.

Le graphe (3.21) nous indique l'influence de la profondeur maximale sur les performances du système. A la différence du problème précédent, ce graphe nous indique qu'en augmentant la profondeur maximale des arbres de « 5 » à « 10 », les performances sont clairement améliorées. Cela signifie que les arbres de profondeurs 5 sont généralement insuffisants pour exprimer une solution satisfaisante au problème posé.

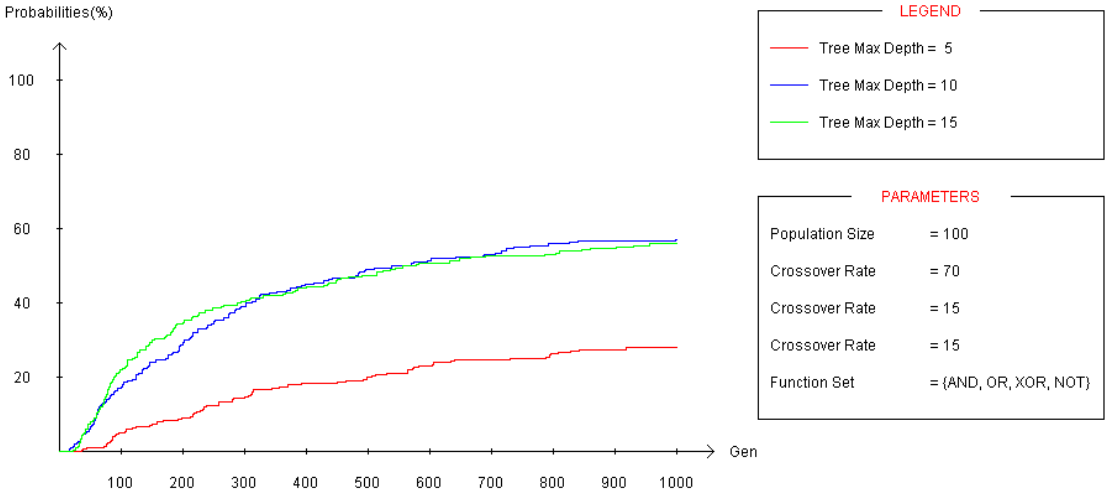


Figure 3.21 – L'influence de la profondeur maximale des arbres sur les performances du système.

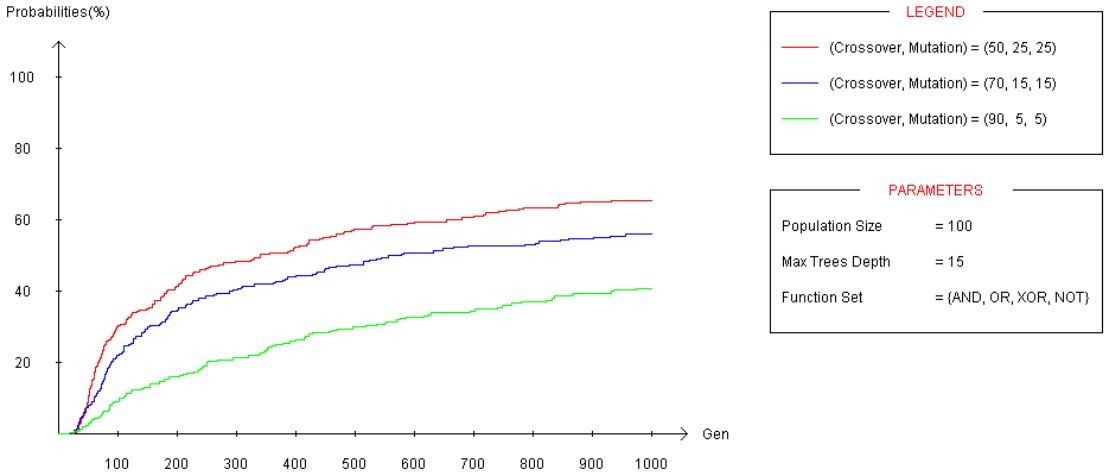


Figure 3.22 –L'influence des opérateurs génétiques sur les performances du système.

La figure (3.22) indique l'influence des opérateurs génétiques sur les performances du système. Nous remarquons que plus le taux de cross-over diminue et que le taux de mutation augmente, plus les performances s'améliorent. Cela indique que la population initiale ne contient pas assez de diversité.

Le graphe (3.23) indique l'influence de l'ensemble des fonctions sur les performances du système. Nous remarquons qu'en excluant le « not » de l'ensemble

des fonctions, aucune solution n'est obtenue. Cela indique que le système n'a pas pu exprimer les solutions au problème avec cet ensemble de fonctions.

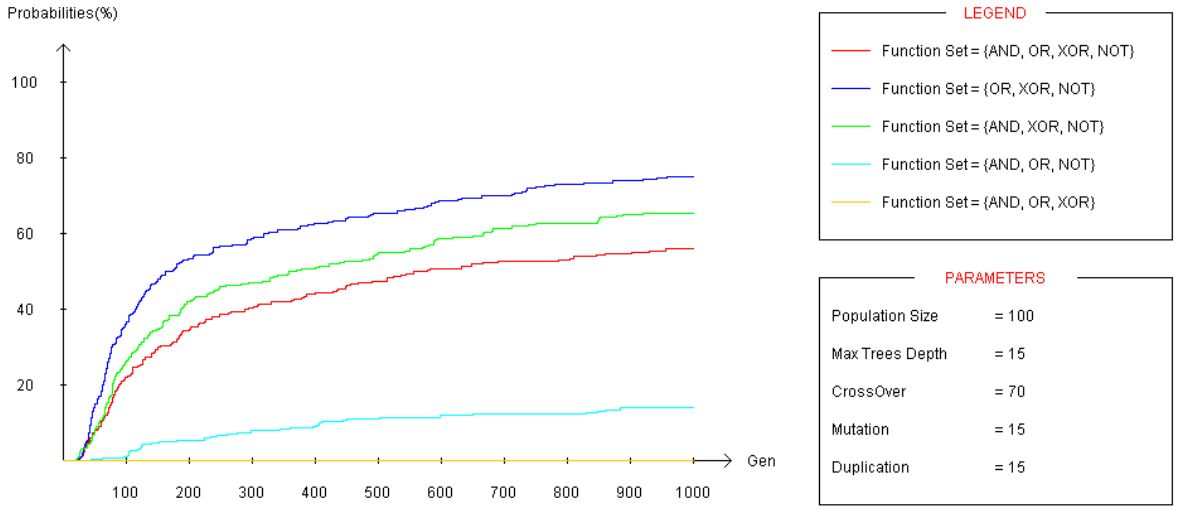


Figure 3.23 – L'influence de l'ensemble des fonctions sur les performances du système.

e- Conclusion

Notre deuxième application montre une méthode simple pour générer des circuits logiques à partir de leurs tables logiques. Il est certes possible d'obtenir des résultats meilleurs en utilisant d'autres méthodes comme la table de karnaugh par exemple. Cependant, notre but était juste de montre qu'il est possible d'utiliser la programmation génétique dans ce genre de problèmes.

Cette application aurait probablement donné de meilleurs résultats si on lui avait ajouté :

- Un module pour la réutilisation du code : avec ce module, l'application peut trouver la solution plus vite.
- Un module de simplification : pour réduire la taille des tables obtenues.

3- Génération automatique des racines de polynômes

La génération automatique de racines symboliques de polynômes consiste étant donné un polynôme de la forme (3.3), à trouver une solution de la forme (3.4).

$$P(x) = A_n x^n + A_{n-1} x^{n-1} + A_{n-2} x^{n-2} \ldots + A_o x \quad (3.3)$$

$$Racine = F(A_n, A_{n-1}, A_{n-2}, \ldots, A_o) \quad (3.4)$$

Les coefficients A1, A2, …, An ne sont pas obligés tous d'êtres symboliques. Si ces derniers sont tous numériques, nous aurons une solution numérique.

a- Primitives

Nous pouvons utiliser les primitives suivantes :

- **Ensemble des terminaux** : Les variables (A1, A2, …, An), ainsi que la constante éphémère R.
- **Ensemble des fonctions** : +, -, x, /, Sqr, Sqrt, Sqr3, Sqrt3, Sqr4, Sqrt4, Sqr5, Sqrt5. Nous pouvons ajouter d'autres fonctions si nous le voulons.

b- Fitness

Soit à calculer la fitness d'une structure évolutionnaire représentant une racine d'un polynôme donné. Le polynôme et la racine contiennent des constantes symboliques. Notre fitness doit être nulle pour une racine 100% exacte. Plus la

structure est fausse, plus la valeur de la fitness doit être plus grande. Afin de calculer cette fitness, nous pouvons suivre les étapes suivantes :

1. Générer une combinaison aléatoire des coefficients A_n, A_{n-1}, ..., A_0.
2. Remplacer les valeurs de ces coefficients dans le polynôme. Nous obtenons donc un polynôme avec des coefficients réels.
3. Remplacer les valeurs des coefficients dans la formule de la racine. Nous obtenons un nombre réel.
4. Calculer la valeur du polynôme pour la racine obtenue.
5. Cette valeur correspond à une fitness partielle.
6. Répéter cette opération « n » fois. La fitness sera la moyenne de ces fitness partielles.

Il est à noter que remplacer une combinaison aléatoire de coefficients dans la formule de la racine peut ne pas être définie a cause d'une opération indéfinie comme une division par zéro. Dans ce cas, nous pouvons tout simplement ignorer cette combinaison. Si toutes les combinaisons donnent des résultats indéfinis, nous induisons que la formule en question est invalide et nous la supprimons de la population.

Nous allons illustrer le calcul de la fitness par un exemple simple. Nous avons un polynôme et deux racines. Nous allons calculer la fitness de chaque racine.

$$P(x) = Ax + B$$

$$R1 = -\frac{B}{A}$$

$$R2 = A + B$$

Figure 3.24 – Exemple de polynôme symbolique ainsi deux racines candidates.

A	B	R1	R2	Polynôme	Fit. Par 1	Fit. Par 2
1	1	-1	2	P(x) = x + 1	0	3
2	2	-1	4	P(x) = 2x + 2	0	10
-3	5	1.6666	2	P(x) = -3x + 5	0	-1
5	0	0	5	P(x) = 5x	0	25
0	-4	NaN	-4	P(x) = -4	NaN	-4
5	-1	0.2	4	P(x) = 5x - 1	0	19
Fitness moyenne					0	8.666

Tableau 3.3 – Calcul de la fitness des deux racines candidates de la figure (3.24)

Nous remarquons que la première racine a une fitness de 0. Cette raine est donc 100% correct. La deuxième formule ne représente donc pas une racine du polynôme étudié.

c- Exemple 1

Soit à trouver une racine pour le polynôme de la figure (3.25).En utilisant les paramètres montrés dans la figure (3.26), nous avons obtenu la solution exprimée par la formule de la figure (3.27).

$$P(x) = Ax+B$$

Figure 3.25 – Polynôme du premier degré.

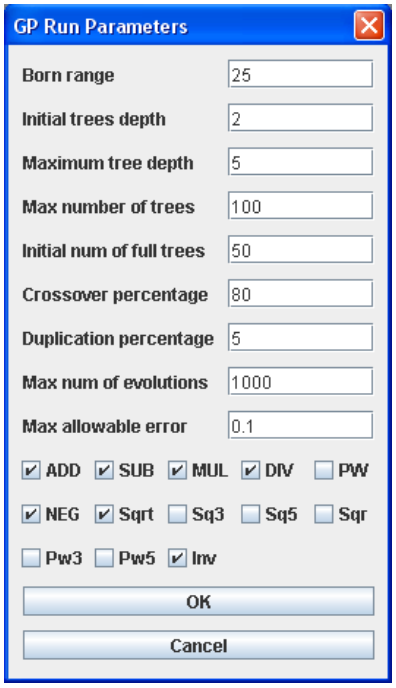


Figure 3.26 – Paramètres utilisés pour induire la racine du polynôme de la figure (3.25).

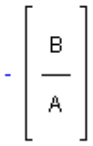


Figure 3.27 – Solution trouvée pour le polynôme de la figure (3.25).

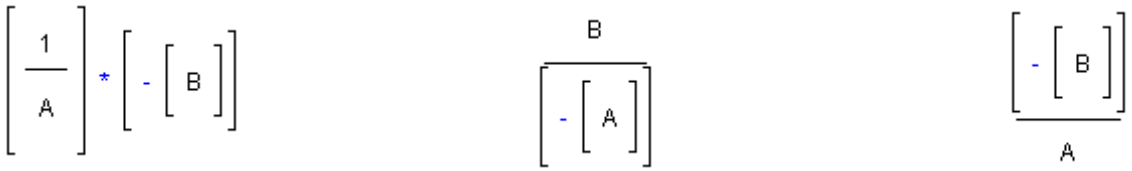


Figure 3.28 – Autres solutions pour le problème de la figure (3.25).

d- Exemple 2

$$P(x) = 2.0Ax^2+2.0B$$

Figure 3.29 – Equation du second ordre

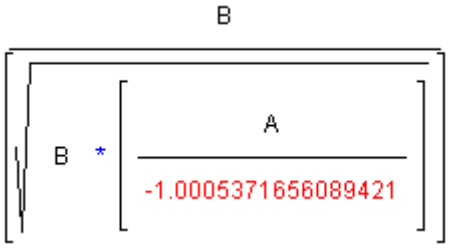


Figure 3.30 – Solution trouvée pour l'équation de la figure (3.26).

Dans la figure (3.30), la solution exprimée n'est qu'une solution approchée (l'erreur est de 0.012). Cette solution aurait était parfaite si la constante en rouge serait « 1 ».

e- Etude de performances

La figure (3.31) ainsi que le tableau (3.3) montrent les performances du système pour les équations du premier degré et les équations du deuxième degré.

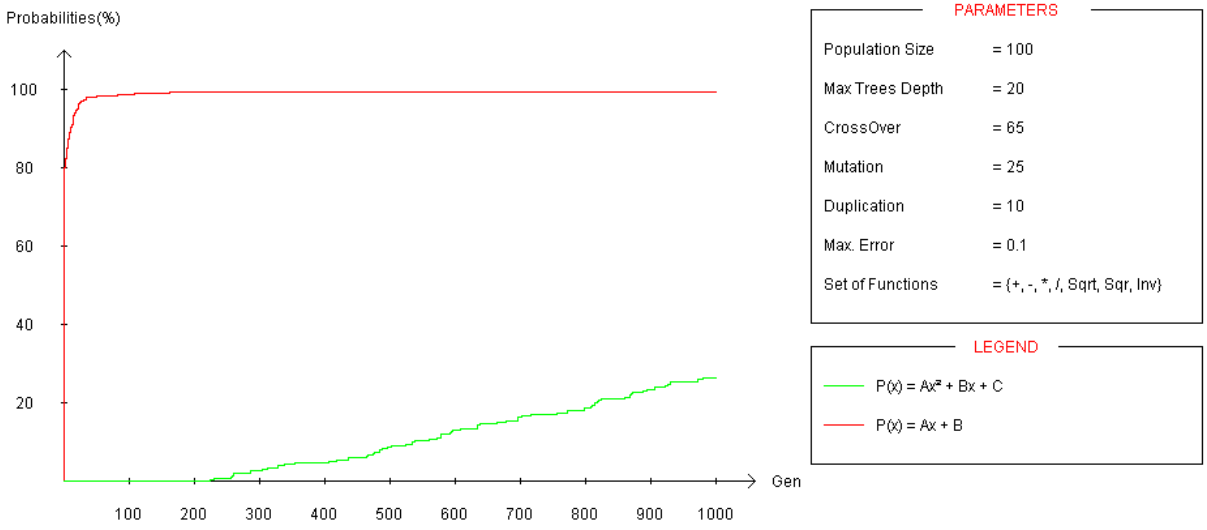


Figure 3.31 – Probabilités de convergence du système pour les équations du premier et second ordre.

Générations	Taux de class (%) Equation du 1ère ordre	Taux de class (%) Equation du 2ème ordre
0	0	0
100	98.8	0
200	99.6	0
300	99.6	2.8
400	99.6	4.8
500	99.6	8.8
600	99.6	13.2
700	99.6	16.4
800	99.6	18.8
900	99.6	23.6
1000	99.6	26.4

Tableau 3.4 – Taux de classification par génération pour les équations de première et deuxième ordre.

f- Conclusion

La résolution symbolique des équations mathématiques est l'une des applications les plus intéressantes de l'intelligence artificielle. A la différence des deux problèmes précédents, il n'existe pas de méthode universelle pour résoudre des équations symboliques. Nous avons montré à travers notre troisième application, une méthode simple pour résoudre ce problème dans le cas des polynômes.

Tout comme les deux applications précédentes, cette application peut être améliorée en lui ajoutant :

- Un module qui gère la réutilisation du code.
- Un module de simplification car les racines obtenues sont souvent énormes.

4- Conclusion

Dans ce chapitre, nous avons montré l'utilité de la programmation génétique à travers trois applications simples. Ces dernières nous ont permis de montré l'influence des différents paramètres des systèmes génétiques, à savoir : la taille de la population, l'ensemble des fonctions, le taux de cross-over, le taux de mutation ainsi que le nombre maximal de générations.

Conclusion et perspectives

Aujourd'hui, la programmation génétique est un domaine de recherche intéressant et très actif. En deux décennies, cette technique polyvalente a prouvé sa capacité à résoudre un nombre impressionnant de problèmes difficiles en donnant des solutions non optimales mais satisfaisantes. Même si les premiers travaux concernant la programmation génétique ont débutés il y a 20 ans, beaucoup reste à découvrir dans ce domaine.

Dans ce mémoire, nous avons abordé la programmation génétique en exposant ses concepts fondamentaux ainsi que certains concepts avancés. Nous avons aussi présenté trois applications simples de la programmation génétique. Notre étude nous a permis de démontrer la capacité de la programmation génétique à trouver des solutions satisfaisantes à plusieurs types de problèmes. Notre étude nous a aussi permis de connaitre les limitations de la programmation génétique. En effet, cette technique souffre d'un certain nombre de problèmes. Par exemple les solutions qu'elles renvoi ne sont pas les même à chaque fois et ne sont que rarement optimales. Il existe aussi les problèmes de l'explosion de la taille des arbres générés ainsi que celui du temps de convergence qui est parfois énorme.

Ce travail peut être complété en abordant « les méthodes de réutilisation de code », « les méthodes de contrainte de structures » ainsi que « les opérateurs génétiques alternatifs ».

Références

[Wolfgang - 1998] - Wolfgang Banzhaf et al, « Genetic Programming An Introduction On the Automatic Evolution of Computer Programs and its Application », Morgan Kaufmann, 1998.

[Riccardo Poli - 2008] – Riccardo Poli , William B. Langdon , Nicholas F. McPhee , « A Field Guide to Genetic Programming», 2008.

[Robilliard] – Denis Robilliard, Cyril Fonlupt LIL – Laboratoire d'Informatique du Littoral, Calais ULCO – Université du Littoral-Côte d'Opale.

[KOZA - 1992] – John R. Koza , « Genetic Programming - On the Programming of Computers by Means of Natural Selection», A Bradford Book The MIT Press, 1992.

[Grefenstette and Baker, 1989] - Grefenstette, J. J. and Baker, J. E. (1989). How genetic algorithms work: A critical look at implicit parallelism. In Proc. 3rd International Conference on Genetic Algorithms, pages 20-27, San Mateo, CA. Morgan Kaufmann, San Francisco, CA.

[Whitley, 1989] - Whitley, D. (1989). The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In Schaffer, J. D., editor, Proc. 3rd Int. Conference on Genetic Algorithms, pages 116-121, San Mateo, CA. Morgan Kaufmann,
San Francisco, CA.

[R. Aler, D. Borrajo, and P. Isasi] - R. Aler, D. Borrajo, and P. Isasi. Using genetic programming to learn and improve control knowledge. Artificial Intelligence, 141(1-2):29–56, October 2002. URL http: //scalab.uc3m.es/~dborrajo/papers/aij-evock.ps.gz.

[Holmes - 1995] - P. Holmes. The odin genetic programming system. Tech Report RR-95-3, Computer Studies, Napier University, Craiglockhart, 216 Colinton Road, Edinburgh, EH14 1DJ, 1995. URL http://citeseer.ist.psu.edu/holmes95odin.html.

[Hsu and S. M. Gustafson – 2001] - W. H. Hsu and S. M. Gustafson. Wrappers for automatic parameter tuning in multiagent optimization by genetic programming. In IJCAI-2001 Workshop on Wrappers for Performance Enhancement in Knowledge Discovery in Databases (KDD), Seattle, Washington, USA, 4 August 2001.

[Teller and Veloso, 1995b] - Teller, A. and Veloso, M. (1995b). PADO: Learning tree structured algorithms for orchestration into an object recognition system. Technical Report CMU-CS-95- 101, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

[CREPEAU-1995] - R. L. Crepeau. Genetic evolution of machine language software. In J. P. Rosca, editor, Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications, pages 121–134, Tahoe City, California, USA, 9 July 1995. URL http: //www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/GEMS_Article.pdf. GPBiB

[FOSTER - 2001] - J. A. Foster, E. Lutton, J. Miller, C. Ryan, and A. G. B. Tettamanzi, editors. Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002, volume 2278 of LNCS, Kinsale, Ireland, 3-5 April 2002. Springer-Verlag. ISBN 3-540-43378-3. GPBiB

[Altenberg, 1995] - Altenberg, L. (1995). Genome growth and the evolution of the genotype-phenotype map. In Banzhaf, W. and Eeckman, F. H., editors, Evolution as a Computational Process. Springer-Verlag, Berlin, Germany.

[Tackett - 1994] - Tackett, W. A. (1994). Recombination, Selection, and the Genetic Construction of Computer Programs.

[Nordin et al., 1996] - Nordin, P., Francone, F., and Banzhaf, W. (1996). Explicitly defined introns and destructive crossover in genetic programming. In Angeline, P. J. and Kinnear, Jr., K. E., editors, Advances in Genetic Programming 2, chapter 6, pages 111- 134. MIT Press, Cambridge, MA.

[W. B. Langdon] - W. B. Langdon. The evolution of size in variable length representations. In 1998 IEEE International Conference on Evolutionary

Computation, pages 633–638, Anchorage, Alaska, USA, 5-9 May 1998. IEEE Press. URL : http://www.cs.bham.ac.uk/~wbl/ftp/papers/WBL.wcci98_bloat.pdf.

[Kinnear-1993] - K. E. Kinnear, Jr. Evolving a sort: Lessons in genetic programming. In Proceedings of the 1993 International Conference on Neural Networks, volume 2, pages 881– 888, San Francisco, USA, 28 March-1 April 1993. IEEE Press. ISBN 0-7803-0999-5. URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/ftp.io.com/papers/kinnear.icnn93.ps.Z.

[NORDIN-1994] - P. Nordin. A compiling genetic programming system that directly manipulates the machine code. In K. E. Kinnear, Jr., editor, Advances in Genetic Programming, chapter 14, pages 311–331. MIT Press, 1994. URL http://cognet.mit.edu/library/books/ view?isbn=0262111888.

[DEV-ALGO] - http://rperrot.developpez.com/articles/algo/structures/arbres/

[GENSITE] - http://www.geneticprogramming.com/Tutorial/

[GENSITE2] - http://www.geneticprogramming.us/What_is_Genetic_Programming.html

[WIKI-ARBRES] - http://fr.wikipedia.org/wiki/Arbre_binaire#Parcours_pr.C3.A9fixe.2C_infixe_et_postfixe

Résumé

La programmation génétique est une technique évolutionnaire qui induit des structures exécutables pour résoudre des problèmes difficiles. Cette technique fait partie des algorithmes dits évolutionnaires tout comme les algorithmes génétiques.

Dans ce travail, nous avons présenté les concepts fondamentaux ainsi que certains concepts avancés de la programmation génétique. Nous avons ensuite implémenté trois applications qui utilisent la programmation génétique afin de traiter les problèmes de « génération automatique de modèles mathématiques », « génération automatique de circuits logiques» ainsi que « la résolution symbolique des polynômes ». Pour ainsi appuyer l'importance de cette technique, une étude des performances a été implémentée pour chaque application afin de montrer la force et les faiblesses de la programmation génétique.

Mots clés : Programmation génétique, Algorithmes évolutionnaires.

Abstract

Genetic programming is an evolutionary technique that evolves executable structures to solve difficult problems. This technique is part of evolutionary algorithms just like genetic algorithms.

In this work, we have presented the fundamentals as well as some advanced concepts of genetic programming. We have also implemented three programs that use genetic programming to solve the problems of "modeling through regressions", 'digital circuits generation", as well as "symbolic resolution of polynomials". For each application, we have done some performances studies in order to show the strength and the weaknesses of the genetic programming technique.

Key words: Genetic Programming, Evolutionary Algorithms.

ملخص

البرمجة الجينية هي تقنية تطورية تنتج البرامج الحاسوبية لحل المشاكل الصعبة. ه ذه التقنية تنتمي لخوارزميات التطورية مثل الخوارزميات الجينية.

في هذه المذكرة, لقد قمنا بعرض أهم المبادئ الأساسية و بعض المبادئ المتقدمة الخاصة بالبرمجة الجينية. لقد قمنا كذلك بتصميم ثلاثة برامج حاسوبية تستعمل البرمجة الجينية لحل المشاكل التالية : "البحث عن الدوال الرياضية", "التصميم الذاتي للدوائر المنطقية" و "البحث عن الجذور الرمزية و الرقمية لكثيرات الحدود". مع برنامج, لقد قمنا كذلك بدراسة أداك كل برنامج بهدف عرض نقاط القوة و الضعف هده التقنية.

الكلمات المفتاحية : البرمجة الجينية, الخوارزميات الجينية.