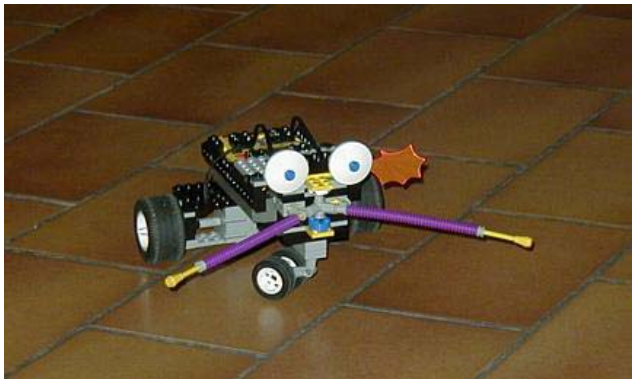
PROJET D'INTELLIGENCE ARTIFICIELLE



RECHERCHE DE PROGRAMMES PAR ALGORITHMES GÉNÉTIQUES

Table des matières

Dossier

I. Présentation générale

I.1. Le langage utilisé

I.1.1. Les langages envisagés

Les langages envisagés (en fonction de nos connaissances et de leur affinité avec le domaine de l'intelligence artificielle) sont :

- C / C++ ;
- Java ;
- OCaml ;
- Prolog ;
- Scheme.

I.1.2. Le langage choisi

Pour choisir un langage, il faut d'abord savoir quelles sont les qualités que celui-ci doit posséder.

Dans ce projet, nous voulons implémenter un algorithme génétique pour trouver un programme valide de contrôle de robot. Cela suppose plusieurs contraintes au niveau du langage :

1. le programme compilé ou interprété doit être rapide car un algorithme génétique nécessite beaucoup de calcul pour évoluer ;
2. le langage doit faciliter la manipulation des arbres car le programme de contrôle recherché sera construit sous la forme d'un arbre ;
3. le langage doit permettre une gestion performante de la mémoire puisque celle-ci est en évolution permanente.

Voyons comment, les langages envisagés se comportent face à ces critères :

	1	2	3
C	+++	–	++ / ––
Java	=	–	+
OCaml	++ / +	++	+
Prolog	––	++	?
Scheme	–– (?)	++	?

Rem : Deux remarques à propos du tableau ci-dessus :
1/ Il n'y a pas mieux que le C pour gérer la mémoire. Cependant une gestion réellement efficace de la mémoire est très difficile à programmer. D'autre part, si on gère « bêtement » la mémoire à coups répétés de malloc et free alors cela aura de graves conséquences sur les performances du programme.
2/ OCaml dispose d'un compilateur en code byte qui est plus performant que celui de Java et d'un compilateur en code natif dont les performances sont proches d celles du C.

Compte tenu des critères retenus, nous avons choisi OCaml comme langage de programmation.

I.2. Organisation des sources

I.2.1. Découpage

Il est possible de découper le programme en deux parties distinctes :

- une première qui concerne l'interprétation et l'évaluation des programmes créés ;
- une deuxième qui met en place les algorithmes génétiques.

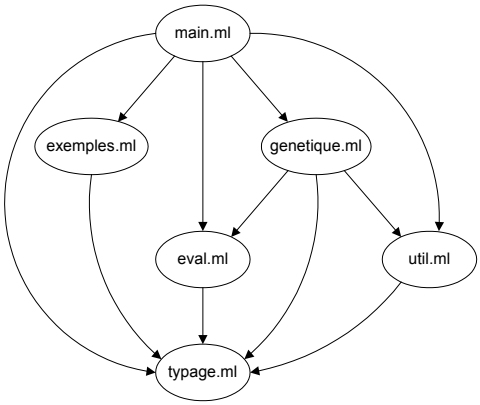
Suivant ce découpage, nous avons réparti le code source du projet en six fichiers :

- Deux fichiers reprenant les deux parties vues ci-dessus et constituant donc le cœur du projet :
  - eval.ml qui met en place toutes les fonctions permettant d'évaluer un programme ;
  - genetique.ml contenant les fonctions servant à créer la première génération de programmes et à passer d'une génération à la suivante.
- Trois fichiers permettant d'utiliser ces fonctions :
  - typage.ml définissant les types et structures utilisés ;
  - util.ml implémentant quelques fonctions utiles annexes comme l'affichage sous forme lisible d'un programme ;
  - main.ml qui contient la fonction appelée au lancement de l'application.
- Un fichier d'exemples pour tester les fonctions codées : exemples.ml.

A tous ces fichiers, il faut ajouter un fichier de configuration (`configuration`) qui contient la grille et quelques paramètres des algorithmes génétiques.

I.2.2. Hiérarchie

Voici le graphe de dépendance des fichiers :



II. Implémentation

II.1. Les types et structures

II.1.1. Structure des arbres-programmes

Les programmes sont représentés par des arbres, type `arbre-prog`, dont les nœuds sont des enregistrements (au sens OCaml) dont les champs sont (ie. les nœuds sont étiqueté par) :

- `code` : type `node`, indique l'instruction attachée à ce nœud et éventuellement ses fils ;

- `haut` : type `int`, contient la hauteur du sous-arbre dont le nœud est racine.

Cette structure a été choisie car elle est très facile à manipuler (avec des `match`) dans le reste du projet.

II.1.2. Interprétation d'un nœud

L'interprétation d'un nœud est de type `eval_prog`. Elle peut être soit :

- un booléen (`EvalBoolVrai`, `EvalBoolFaux`) ;
- une action (`EvalActionNord`, `EvalActionOuest`, `EvalActionEst`, `EvalActionSud`) ;
- une non-action (`EvalActionStop`).

II.2. Les règles

II.2.1. Règles d'interprétation

Toutes les combinaisons de nœuds sont possibles. Voyons comment les interpréter.

II.2.1.1. Les actions

Les actions s'interprètent toujours de la même manière.

$$(\text{NodeActNord})\frac{}{\vdash \text{NodeActNord} \rightarrow \text{EvalActionNord}} \qquad (\text{NodeActEst})\frac{}{\vdash \text{NodeActEst} \rightarrow \text{EvalActionEst}}$$

Erreur ! Des objets ne peuvent pas être créés à partir des codes de champs de mise en forme. **Erreur ! Des objets ne peuvent pas être créés à partir des codes de champs de mise en forme.**

II.2.1.2. Les constantes

Les constantes s'interprètent toujours de la même manière.

$$(\text{NodeConst0})\frac{}{\vdash \text{NodeConst0} \rightarrow \text{EvalBoolFaux}} \qquad (\text{NodeConst1})\frac{}{\vdash \text{NodeConst1} \rightarrow \text{EvalBoolVrai}}$$

II.2.1.3. Les variables

Les variables de nos programmes sont les nœuds référant aux capteurs. L'état des capteurs est mémorisé dans un tableau de booléens que nous appèlerons `capteurs`.

$$(\text{NodeVarN})\frac{E \vdash \text{capteurs.}(0)=\text{true}}{E \vdash \text{NodeVarN} \rightarrow \text{EvalBoolVrai}} \qquad (\text{NodeVarN})\frac{E \vdash \text{capteurs.}(0)=\text{false}}{E \vdash \text{NodeVarN} \rightarrow \text{EvalBoolFaux}}$$

$$(\text{NodeVarNO})\frac{E \vdash \text{capteurs.}(1)=\text{true}}{E \vdash \text{NodeVarNO} \rightarrow \text{EvalBoolVrai}} \qquad (\text{NodeVarNO})\frac{E \vdash \text{capteurs.}(1)=\text{false}}{E \vdash \text{NodeVarNO} \rightarrow \text{EvalBoolFaux}}$$

… …

$$(\text{NodeVarS})\frac{E \vdash \text{capteurs.}(7)=\text{true}}{E \vdash \text{NodeVarS} \rightarrow \text{EvalBoolVrai}} \qquad (\text{NodeVarS})\frac{E \vdash \text{capteurs.}(7)=\text{false}}{E \vdash \text{NodeVarS} \rightarrow \text{EvalBoolFaux}}$$

II.2.1.4. Les opérateurs

L'interprétation des opérateurs dépend de l'interprétation de leur fils.

II.2.1.4.1. L'opérateur conditionnel

$$(\text{NodeOpSI})\frac{E \vdash f1 \rightarrow \text{EvalAction}X}{E \vdash \text{NodeOpSI}(f1,\_,\_) \rightarrow \text{EvalAction}X} \qquad \text{où } X \in \{\text{Stop,Nord,Ouest,Est,Sud}\}$$

$$(\text{NodeOpSI})\frac{E \vdash f1 \rightarrow \text{EvalBoolVrai} \wedge (E \vdash f2 \rightarrow \text{EvalBoolVrai} \vee E \vdash f2 \rightarrow \text{EvalBoolFaux})}{E \vdash \text{NodeOpSI}(f1,f2,\_) \rightarrow \text{EvalActionStop}}$$

$$(\text{NodeOpSI})\frac{E \vdash f1 \to \text{EvalBoolVrai} \wedge E \vdash f2 \to \text{EvalAction}X}{E \vdash \text{NodeOpSI}(f1,f2,\_) \to \text{EvalAction}X} \text{ où } X \in \{\text{Stop,Nord,Ouest,Est,Sud}\}$$

$$(\text{NodeOpSI})\frac{E \vdash f1 \to \text{EvalBoolFaux} \wedge (E \vdash f3 \to \text{EvalBoolVrai} \vee E \vdash f3 \to \text{EvalBoolFaux})}{E \vdash \text{NodeOpSI}(f1,\_,f3) \to \text{EvalActionStop}}$$

$$(\text{NodeOpSI})\frac{E \vdash f1 \to \text{EvalBoolFaux} \wedge E \vdash f3 \to \text{EvalAction}X}{E \vdash \text{NodeOpSI}(f1,\_,f3) \to \text{EvalAction}X} \text{ où } X \in \{\text{Stop,Nord,Ouest,Est,Sud}\}$$

II.2.1.4.2. L'opérateur conjonctif

$$(\text{NodeOpET})\frac{E \vdash f1 \to \text{EvalAction}X}{E \vdash \text{NodeOpET}(f1,\_) \to \text{EvalAction}X} \text{ où } X \in \{\text{Stop,Nord,Ouest,Est,Sud}\}$$

$$(\text{NodeOpET})\frac{E \vdash f1 \to \text{EvalBoolFaux}}{E \vdash \text{NodeOpET}(f1,\_) \to \text{EvalBoolFaux}}$$

$$(\text{NodeOpET})\frac{E \vdash f1 \to \text{EvalBoolVrai} \wedge E \vdash f2 \to X}{E \vdash \text{NodeOpET}(f1,f2) \to X}$$

II.2.1.4.3. L'opérateur disjonctif

$$(\text{NodeOpOU})\frac{E \vdash f1 \to \text{EvalAction}X}{E \vdash \text{NodeOpOU}(f1,\_) \to \text{EvalAction}X} \text{ où } X \in \{\text{Stop,Nord,Ouest,Est,Sud}\}$$

$$(\text{NodeOpOU})\frac{E \vdash f1 \to \text{EvalBoolVrai}}{E \vdash \text{NodeOpOU}(f1,\_) \to \text{EvalBoolVrai}}$$

$$(\text{NodeOpOU})\frac{E \vdash f1 \to \text{EvalBoolFaux} \wedge E \vdash f2 \to X}{E \vdash \text{NodeOpOU}(f1,f2) \to X}$$

II.2.1.4.4. L'opérateur négatif

$$(\text{NodeOpNON})\frac{E \vdash f1 \to \text{EvalAction}X}{E \vdash \text{NodeOpNON}(f1) \to \text{EvalAction}X} \text{ où } X \in \{\text{Stop,Nord,Ouest,Est,Sud}\}$$

$$(\text{NodeOpNON})\frac{E \vdash f1 \to \text{EvalBoolVrai}}{E \vdash \text{NodeOpNON}(f1) \to \text{EvalBoolFaux}} \quad (\text{NodeOpNON})\frac{E \vdash f1 \to \text{EvalBoolFaux}}{E \vdash \text{NodeOpNON}(f1) \to \text{EvalBoolVrai}}$$

II.2.1.5. Le programme vide

Le programme s'interprète toujours de la même manière.

$$(\text{Empty})\frac{}{\vdash \text{Empty} \to \text{EvalActionStop}}$$

II.2.2. Règles de simplification

Lorsque le taille moyenne des arbre-programmes atteint un certain seuil (réglable dans le fichier de configuration), il est utile de réduire leur taille. Pour cela, on simplifie les programmes avec les règles suivantes.

II.2.2.1. Les nœuds sans fils

Les nœuds sans fils ne peuvent pas être simplifiés.

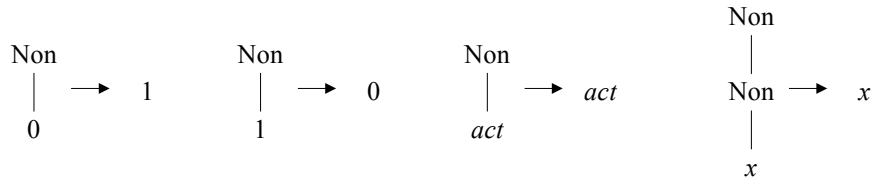
II.2.2.2. Les nœuds avec fils

On utilise la convention suivante dans les schémas ci-dessous :

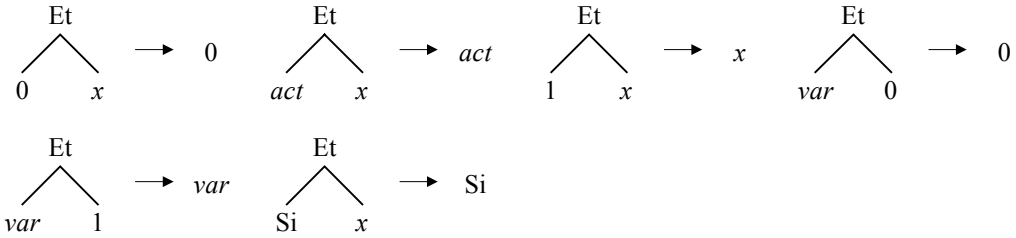
- *x* et *y* sont des nœuds quelconques ;
- *act* est un nœud se simplifiant en NodeAct*X* où $X \in \{\text{Stop,Nord,Ouest,Est,Sud}\}$;
- *var* est un nœud se simplifiant en NodeVar*X* où $X \in \{\text{N,NO,NE,O,E,SO,SE,S}\}$.

Les autres conventions (Non, Et, …, 0 et 1) se comprennent facilement.

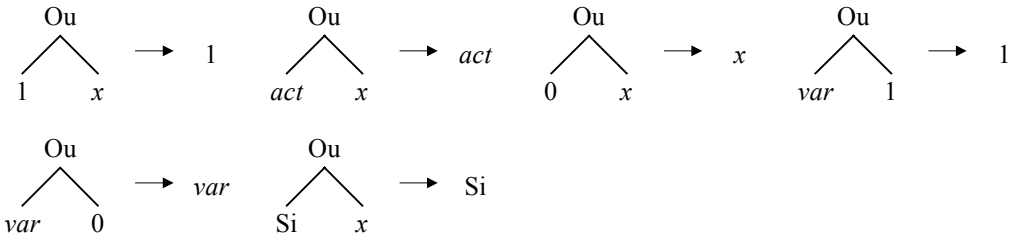
II.2.2.2.1. La négation



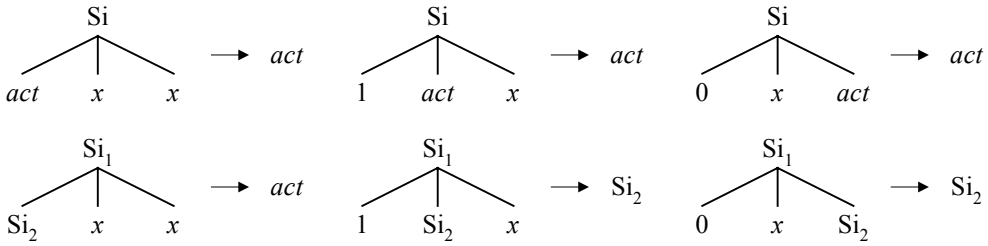
II.2.2.2.2. La conjonction



II.2.2.2.3. La disjonction



II.2.2.2.4. La condition



II.3. Les algorithmes

II.3.1. Génération de la première population

La première population est générée de manière probabiliste. Donc on peut décomposer cette création en deux étapes :

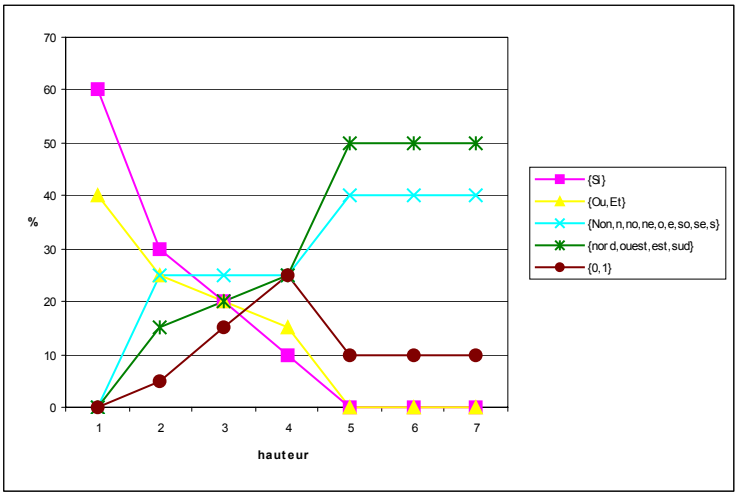
- mise en place des probabilités d'obtenir un certain type de nœud à une hauteur donnée ;
- création aléatoire d'un programme suivant ces probabilités.

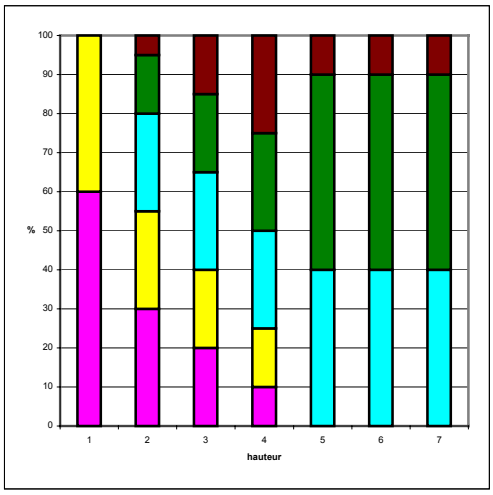
II.3.1.1. Mise en place des probabilités

Dans le fichier de configuration il est possible de choisir entre deux manières de calculer les probabilités.

La manière la plus simple mais aussi la moins efficace est d'utiliser des probabilités équiprobables (fonction `make_proba1`), c'est à dire qu'à toutes les hauteurs, exceptée la dernière, on a autant de chance d'obtenir un `NodeOpSI` ou un `NodeConst0` par exemple. A la dernière hauteur (hauteur fixée dans le fichier de configuration) il y a 0% de chance d'obtenir un nœud ayant au moins un fils.

Une manière plus efficace consiste à faire en sorte que par exemple le `NodeOpSI` est beaucoup de chance d'être tiré à des hauteurs proches de la racine et très peu plus on se rapproche de la hauteur maximale (aucune à la hauteur maximale). C'est la fonction `make_proba_2`. Voici par exemple les probabilités calculées pour une hauteur « maximum » de 5 (en fait il est possible de créer des arbres dont la hauteur est supérieure) :





Une fois calculées les probabilités sont stockées pour chaque hauteur dans un tableau de 1+18 entrées (il y a 18 types de nœuds différents) et chacun de ces tableaux est mis dans un autre tableau de « hauteur maximum » entrées. Voici pour l'exemple précédent le tableau pour la hauteur 4 :

type nœud		Si	Et	Ou	Non	n	no	ne	o	e	so	se	s	nord	ouest	est	sud	0	1
indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
tab des probas :	0	100	175	250	278	306	337	365	393	421	449	477	500	562	623	685	750	875	1000

II.3.1.2. Création d'un programme

La création d'un programme (fonction `prog_aleat`) consiste simplement à tirer au hasard un nombre entre 0 et 999 (les probabilités sont exprimées en $^0/_{00}$) et à voir, dans un des tableaux de probabilités calculés précédemment, à quel type de nœud ce nombre correspond. Cette recherche est effectuée de manière efficace (en O(lg n) par dichotomie) par la fonction `quick_find` dans util.ml.

II.3.2. Passage à la population suivante

Le passage d'une génération à la suivante est réalisé comme indiqué dans le sujet (excepté en ce qui concerne les chiffres puisqu'ils sont modifiables dans le fichier de configuration). C'est à dire :

- Certains programmes sont recopiés sans modification. Ces programmes sont dits « immortels ».
- Les autres programmes sont construits par croisement de deux programmes de la génération précédente.

De plus, il est possible de compiler le projet de sorte qu'il utilise ou non la mutation.

II.3.2.1. Choix d'un sous-arbre

Lors d'un croisement entre deux programmes *P1* et *P2*, il faut choisir un sous-arbre *p* de *P1* à supprimer et un sous-arbre dans *P2* à greffer à la place de *p*.

II.3.2.1.1. A greffer

Le choix du sous-arbre de *P2* est fait grâce à la fonction `sous_prog`. Ce choix se fait en trois étapes :

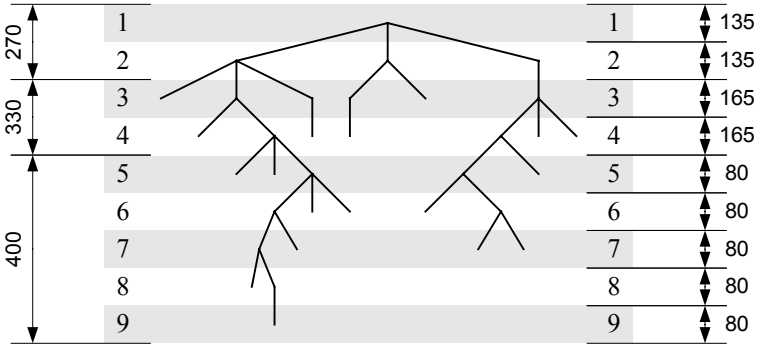
1. tirage (selon des probabilités empiriques) d'une profondeur de l'arbre ;
2. recherche de tous les nœuds se trouvant à cette profondeur ;

3. tirage d'un nœud parmi ceux-ci en affectant des probabilités proportionnelles à la hauteur des arbres dont ces nœuds sont racines.

Afin d'obtenir le plus souvent des sous-arbres de hauteur pas trop petite, on cherche à favoriser à l'étape 1 le tirage d'une profondeur proche de la racine. Pour cela, si H est la hauteur de *P2*, on affecte :

- 270 chances sur 1000 de choisir une profondeur $\pi < H / 4$;
- 330 chances sur 1000 de choisir une profondeur $H / 4 \leq \pi < H / 2$
- 400 chances sur 1000 de choisir une profondeur $H / 2 \leq \pi$

Par exemple :



Rem : Ces probabilité sont mises sous forme de tableau et on utilise `quick_find` dans util.ml et un nombre aléatoire entre 0 et 999 pour le tirage.

II.3.2.1.2. A éliminer

Le choix d'un sous-arbre à éliminer dans *P1* est fait dans les fonctions `croisement` (sans mutation) ou `mutation` (avec mutation) par un algorithme glouton probabiliste. Le principe de cet algorithme est de parcourir *P1* depuis sa racine et à chaque nœud par lequel on passe :

- Soit ce nœud n'a pas de fils et dans ce cas on greffe un sous-arbre de *P2* à la place (en appelant la fonction `sous_prog`).
- Soit ce nœud a un ou plusieurs fils, alors on tire au hasard un nombre entre 1 et 999 :
  - si ce nombre est inférieur à une probabilité fixée dans la fonction (250 semble bien marcher) alors on greffe un sous-arbre de *P2* à sa place ;
  - sinon on choisit équiprobablement un des fils dans lequel on va faire la greffe et on recopie les autres.

II.3.2.2. Mutation

La mutation est un algorithme similaire à celui qui choisit un sous-arbre à éliminer. C'est à dire que lorsqu'on recopie un nœud, on teste, selon une probabilité dépendant des générations précédentes, si le nœud mute ou non.

Les fonctions permettant la mutation sont : `mutation`, `cp_mut_prog` et `pop_mut_suivante`.

Les possibilités de mutation sont :

- {Et,Ou} → {Ou,Et}
- {Non} → {Non,∅}
- {Si} → {Si}
- {nord,ouest,est,sud,n,no,ne,o,e,so,se,s,0,1} → { nord,ouest,est,sud,n,no,ne,o,e,so,se,s,0,1}

La probabilité de mutation minimale est de 20 pour mille à quoi on ajoute (on appelle Δ la différence entre le score moyen, arrondi à l'entier inférieur, de la population actuelle et celui de la population précédente) :

- 60 si $\Delta = 0$;
- 15 si $\Delta = 1$;

- 5 si $\Delta = 2$;
- 0 autrement.

Ces valeurs ont été déterminées empiriquement par une série de tests.

III. Exemples

III.1. Un exemple

Voici un exemple :

```
Generation  0 : score max= 97 moy=11.0 / haut max= 8 moy=4.3
Generation  1 : score max=133 moy=20.3 / haut max=10 moy=4.6
Generation  2 : score max=151 moy=38.6 / haut max=12 moy=5.2
Generation  3 : score max=160 moy=58.3 / haut max=14 moy=6.0
Generation  4 : score max=272 moy=72.5 / haut max=17 moy=6.4
Generation  5 : score max=275 moy=77.5 / haut max=21 moy=6.9
Generation  6 : score max=320 moy=87.4 / haut max=21 moy=7.4
Programme(s) optimal(s) trouve(s):
-------------------
(SI (se)
    (OU (SI (s)
            (SI (SI (n)
                    (est)
                    (ET (OU (ET (o)
                                (nord))
                            (ouest))
                        (ET (0)
                            (s))))
                (sud)
                (no))
            (ET (sud)
                (n)))
        (SI (no)
            (SI (ET (OU (ne)
                        (e))
                    (SI (nord)
                        (o)
                        (sud)))
                (SI (ET (est)
                        (0))
                    (sud)
                    (OU (nord)
                        (se)))
                (o))
            (SI (s)
                (ET (1)
                    (ouest))
                (sud))))
    (SI (OU (ET (n)
                (1))
            (OU (n)
                (OU (SI (o)
                        (nord)
                        (OU (ouest)
                            (ouest)))
                    (OU (sud)
                        (ouest)))))
        (est)
        (SI (so)
            (OU (ET (o)
                    (0))
                (SI (ET (SI (sud)
                            (so)
                            (sud))
                        (e))
                    (no)
                    (s)))
            (1))))
Verification du programme optimal
OK
```

```
Version simplifiee
(SI (se)
    (SI (s)
        (SI (n)
            (est)
            (ET (OU (ET (o)
                        (nord))
                    (ouest))
                (0)))
        (sud))
    (SI (OU (n)
            (OU (n)
                (SI (o)
                    (nord)
                    (ouest))))
        (est)
        (SI (so)
            (sud)
            (1))))
--------------------
(SI (se)
    (OU (SI (s)
            (SI (SI (n)
                    (est)
                    (ET (OU (ET (o)
                                (nord))
                            (ouest))
                        (ET (0)
                            (s))))
                (sud)
                (no))
            (ET (sud)
                (n)))
        (se))
    (SI (n)
        (est)
        (ET (OU (ET (o)
                    (nord))
                (ouest))
            (nord))))
Verification du programme optimal
OK
Version simplifiee
(SI (se)
    (SI (s)
        (SI (n)
            (est)
            (ET (OU (ET (o)
                        (nord))
                    (ouest))
                (0)))
        (sud))
    (SI (n)
        (est)
        (ET (OU (ET (o)
                    (nord))
                (ouest))
            (nord))))
```

On peut voir dans l'exemple précédent qu'à la sixième génération, deux programmes optimaux ont été trouvés. Tous deux sont affiché dans leur version originale et dans une version simplifiée.

La génération initiale est la génération 0 et pour chaque génération sont affichés d'une part le score du meilleur programme et le score moyen pour tous les programmes ; d'autre part la hauteur du plus grand programme et la hauteur moyenne pour tous les programmes.

III.2. Quelques chiffres

Sur 90 tests :

- Au moins un programme optimal a été trouvé avant la 80ème génération 84 fois.
- En moyenne les programmes sont trouvés à la 21ème génération.

Sources