Rapport de stage

Lucas Branchu

Avril-Juin 2025

Automata et programmation génétique à base d'automate fini déterministe

Maître de stage : M. Igor Stéphan
Enseignant référent : M. Benoit Da Mota

Contents

1 Remerciements

Je tiens à remercier M. Igor Stéphan pour m'avoir aiguillé, conseillé et encadré pendant le stage, ainsi que d'avoir répondu à mes questions.
Je remercie également mes camarades pour leur aide et leur soutien tout au long du stage.
Je remercie l'Université d'Angers et l'U.F.R. Sciences pour le cadre de travail.

2 Introduction

Dans le cadre de ma première année de Master Informatique, j'ai réalisé un TER au sein du laboratoire LERIA, sous la supervision de M. Stéphan.

Ce projet consiste à enrichir une implémentation existante en C++ du jeu de société **Small Islands** en y intégrant une approche de programmation génétique pour générer des intelligences artificielles (sous forme d'automates) qui peuvent jouer automatiquement. **Small Islands** est un jeu déterministe. Pour plus de détails sur les règles, veuillez trouver en annexes un lien vers une version numérique des règles du jeu.
Le jeu de société original a la particularité de posséder un mode où l'on peut jouer seul, et où l'on affronte une forme d'intelligence artificielle nommée "Alexis" (dont les comportements sont définis par des cartes). L'un des objectifs est de recréer ces Alexis.
Une fois cela fait, il faut appliquer les principes de la programmation génétique : générer une population d'individus (où on peut y intégrer les Alexis), les évaluer, retenir les meilleurs et favoriser leur reproduction davantage que celle des autres, pour enfin générer la prochaine génération. Et ainsi de suite, jusqu'à atteindre des résultats satisfaisants.
Le projet a été programmé en C++, pour pouvoir s'intégrer plus simplement à l'implémentation déjà fournie.

3 Etude bibliographique

3.1 Introduction

La programmation génétique est basée sur les algorithmes évolutionnistes et est inspirée par la théorie de l'évolution, son objectif est de trouver, par itérations successives, les meilleurs programmes pour résoudre au mieux différents problèmes.
Le processus repose sur une approche évolutive : une population initiale de programmes est générée, puis des individus sont sélectionnés sur la base de leurs performances afin d'être recombinés par croisement et mutation. Ce cycle est répété jusqu'à convergence vers une solution satisfaisante.

Les différentes étapes de la programmation génétique sont les suivantes :
- Représentation des données ou des séquences d'actions
- Initialisation de la population (aléatoirement)
- Exécution des programmes (individus)
- Évaluation des programmes (individus)
- Croisement et mutation, création de la prochaine génération

Puis on recommence les phases d'exécution, évaluation, et croisement et mutation, jusqu'à obtenir les résultats souhaités.

La programmation génétique a été formellement introduite au début des années 1990 par *John R. Koza*, chercheur à l'université de Stanford, dans son ouvrage fondateur *Genetic Programming: On the Programming of Computers by Means of Natural Selection (1992)*. Il s'est inspiré du paradigme des algorithmes génétiques de *John Holland (1975)*, mais a adapté ce cadre aux programmes informatiques eux-mêmes.
Koza a montré que des programmes pouvaient être "évolués" automatiquement, selon les principes darwiniens de sélection naturelle, de croisement génétique et de mutation, pour résoudre des problèmes complexes dans des domaines tels que l'optimisation, la robotique, le traitement du signal ou même la conception de circuits électroniques.
Depuis cette époque, la programmation génétique est devenue une sous-discipline active de l'informatique évolutive, avec de nombreuses variantes et domaines d'application.

3.2 Les différentes représentations

La première étape consiste à définir une représentation des individus. Il en existe plusieurs qui ont des résultats différents et représentent des problèmes différents.

3.2.1 Représentation par tableau

La première représentation est par tableau : on y décrit successivement les actions du programme.
Par exemple, dans un labyrinthe où l'on souhaite représenter un individu qui avance (imaginons que 0 = haut, 1 = bas, 2 = gauche, 3 = droite), on peut avoir un tableau de ce genre : [2,2,1,3,1,1,2,1] qui

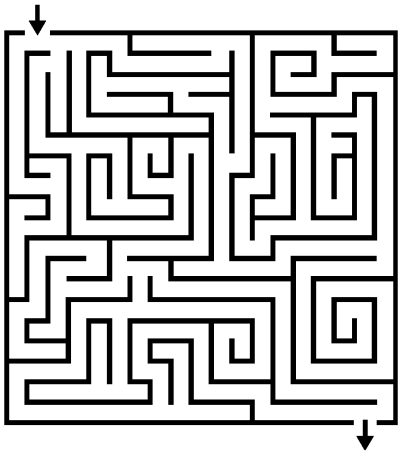


Figure 1: Exemple d'un labyrinthe où le programme essaye en entrant en haut, d'en sortir en bas

représenterait un individu qui va deux fois à gauche, puis en bas, puis à droite, puis deux fois en bas, puis à gauche, et enfin en bas.

3.2.2 Représentation par arbre

Après avoir introduit la représentation par tableau, nous abordons maintenant la représentation arborescente, particulièrement adaptée aux expressions mathématiques
Ici on représente un programme où les feuilles sont les variables et les constantes (appelées *terminaux*), et les noeuds internes sont les opérations arithmétiques (appelés *fonctions*).

Par exemple le programme **max(x+x, x+(3*y))** est représenté de cette manière :

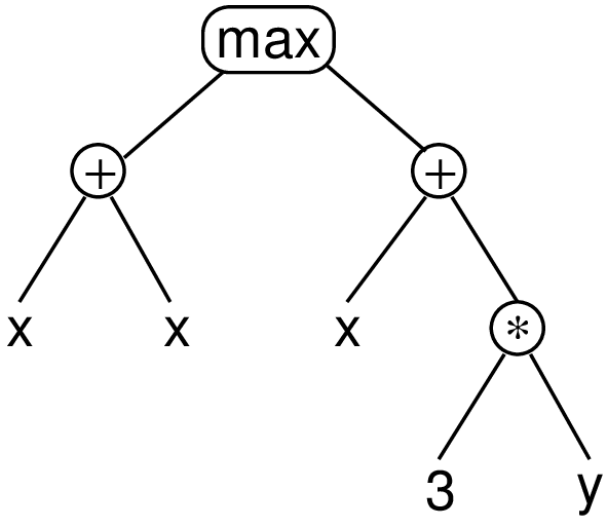


Figure 2: Cité de Poli, Riccardo & Langdon, William & Mcphee, Nicholas. (2008). A Field Guide to Genetic Programming.

3.2.3 Représentation par automate

La représentation par automate consiste à avoir comme transitions les actions effectuées et comme états l'état de la partie.

Pour reprendre l'exemple du labyrinthe, on peut imaginer une représentation et une illustration de la sorte :

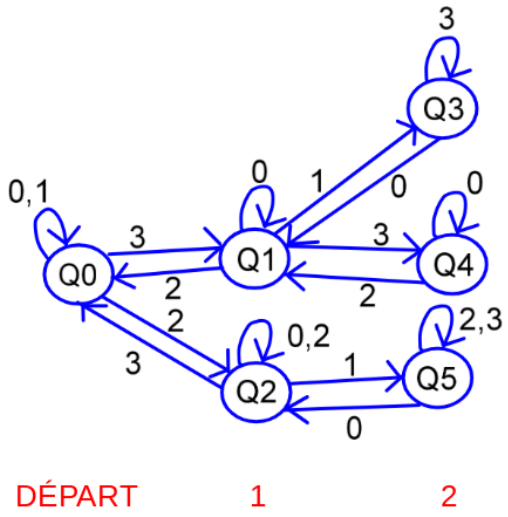


Figure 3: Automate représentant un individu

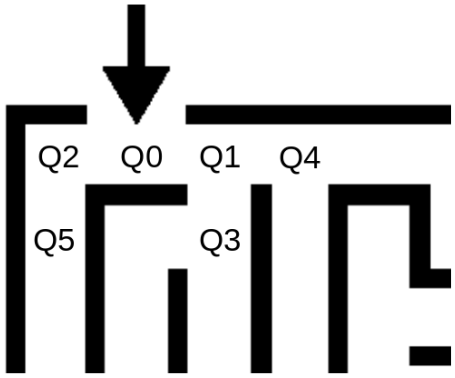


Figure 4: Illustration des actions de l'automate dans le labyrinthe

3.3 La sélection

La sélection est basée sur la fitness de chaque individu. La fitness représente la performance d'un programme face à un problème, dans un jeu cela peut être un nombre de points, dans un labyrinthe cela peut être le nombre de cases parcourues. Une méthode pour sélectionner les individus est de faire des tournois. Les programmes sont comparés selon leur fitness dans un tournoi, et une fois cela fait on obtient un classement de la population, dont le meilleur. De cette manière, on peut sélectionner les meilleurs, ou même le moins bon, selon ce que l'on souhaite obtenir (plus de diversité génétique en gardant des individus moins bons par exemple).

3.4 Les différentes méthodes de croisement

En programmation génétique, le croisement constitue l'un des opérateurs essentiels pour recombiner l'information contenue dans deux individus. Son rôle est d'explorer l'espace de recherche en échangeant des structures intermédiaires qui peuvent porter des propriétés fonctionnelles intéressantes. Il apparaît sous différentes formes selon la représentation choisie pour les individus.

3.4.1 Tableaux

Dans la représentation par tableau, chaque individu est décrit comme une séquence linéaire d'instructions ou d'actions (par exemple, des codes décimaux représentant des directions dans un labyrinthe). Le croisement s'opère en sélectionnant un indice commun (point de croisement) dans les tableaux des deux parents, puis en échangeant les parties situées après cet indice. Cette méthode simple permet une recombinaison directe des séquences d'actions sans altérer la signification locale de chacune d'elles. Elle est particulièrement adaptée aux problèmes où l'ordre et la continuité des actions sont primordiaux pour la solution recherchée.

3.4.2 Subtree crossover

Pour les arbres la méthode subtree crossover est la plus classique. Le croisement s'effectue en sélectionnant aléatoirement un sous-arbre dans chacun des deux parents, puis en échangeant ces sous-structures.
Cette technique est simple à mettre en place, et permet facilement de réutiliser des blocs fonctionnels de programmes.

3.5 Les différentes méthodes de mutation

Alors que le croisement permet de recombiner l'information, la mutation introduit de la diversité en modifiant localement certains individus. Elle joue un rôle important dans l'exploration de l'espace de

recherche et permet d'éviter la stagnation vers des optima locaux. Voici quelques méthodes classiques utilisés sur les arbres :

3.5.1 Subtree mutation

Comme pour la méthode de Subtree Crossover, un sous-arbre est choisi aléatoirement, et puisque c'est une mutation il est cette fois remplacé par un arbre généré aléatoirement.
Le risque est qu'une mutation détruise un sous-arbre qui a le potentiel d'avoir de bonnes performances.

3.5.2 Size-fair subtree mutation

La méthode du Size-fair subtree mutation est similaire au Subtree mutation, à la différence que le sous-arbre généré aléatoirement est de même taille que celui remplacé. Garder une taille raisonnable permet de limiter la complexité.

3.5.3 Node Replacement mutation

La méthode Node Replacement mutation opère une mutation uniquement sur un noeud, en essayant de conserver l'arité. Cela permet de modifier finement le programme sans trop changer la structure globale.

4 Présentation des missions

4.1 Objet et missions du stage

Le but de ce stage est tout d'abord d'automatiser l'implémentation de **Small Islands**, qui pour l'instant fonctionne à partir d'entrées de l'utilisateur dans la console.
Ensuite il faudra implémenter le mode un joueur du jeu de société avec les Alexis, et toutes les règles qui y sont liées.
Puis l'objectif sera d'implémenter la programmation génétique. Tout d'abord en définissant comment seront représentés les Alexis et les intelligences artificielles, puis en implémentant tous les différents algorithmes pour effectuer la programmation génétique sur ces individus.
Enfin, un autre objectif du stage est de comprendre pourquoi les Alexis fonctionnent et sont performants ou non (problème d'équilibrage de jeu).

4.2 Missions réellement réalisées

L'automatisation du jeu, l'implémentation du mode un joueur, et la programmation génétique ont été réalisées.
Par rapport à l'implémentation fournie, un bug qui empêchait les joueurs de marquer correctement leurs points a été corrigé. Un bug provoquant une mauvaise alternance des tours de jeu a été identifié et corrigé. Certaines structures de données ont été modifiées pour pouvoir optimiser le code et l'empêcher de surcharger la mémoire (cette surcharge causait des arrêts du programme lors de la programmation génétique).
Cependant il me manquait du temps pour faire assez de tests sur la fin pour vérifier qu'il ne restait pas quelques bugs dans mon programme (il y en a un qui arrive à peu près 1 partie sur 30 000 et qui peut empêcher un Alexis de placer une maison et donc marquer des points, une mini-correction qui lui fait passer son tour lorsque cela arrive a donc été implémentée), mais cela n'affecte pas les résultats car ce sont des problèmes rares ou mineurs.

5 Développement du jeu automatique

Le développement de l'automatisation du jeu s'est avéré être la partie la plus complexe et chronophage du projet. Tout d'abord, il fallait comprendre l'implémentation du jeu fournie, c'est-à-dire comment les entrées saisies par l'utilisateur étaient utilisées par les différentes méthodes. La logique du jeu dans l'implémentation fournie est à la fois dans une classe Boardgame (qui centralise les opérations sur le plateau du jeu) et une classe Player.
Une sous-classe de Player (Alexis) a été implémentée pour automatiser les actions faites par le Player, elle surcharge les méthodes qui contiennent une partie de la logique du jeu.

5.1 Alexis

Avant d'expliquer l'automatisation du jeu, il est nécessaire de comprendre comment les Alexis fonctionnent dans le jeu de société. Un Alexis est tout d'abord défini par une carte Alexis, cette carte Alexis contient 4 informations : son niveau de difficulté (de 1 à 6), le nombre de cartes "explorer", et le nombre de cartes "aborder" (ce que sont ces cartes sera expliqué juste après), le nombre de points de base (c'est le nombre de points qu'aura Alexis pour la partie peu importe ce qu'il fait), et un nombre de points par maison posée.

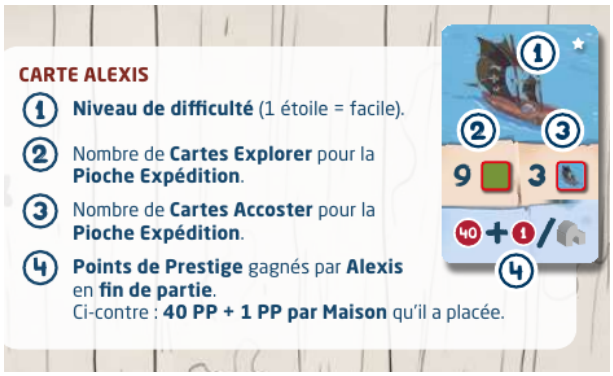


Figure 5: Illustration et explications d'une carte Alexis provenant des règles du jeu

5.2 Carte expédition

Les cartes "explorer" et "aborder" sont des cartes de type expédition. Elles sont au nombre de 15 dans **Small Islands**, elles forment une pioche pour chaque Alexis, une carte est piochée à chaque tour, et elles contiennent diverses informations. L'information importante à retenir, c'est qu'elles vont définir le prochain coup que va jouer Alexis.
J'ai donc implémenté une classe Expedition pour représenter les cartes d'expédition des Alexis.

5.3 Logique du jeu et automatisation

La logique du jeu pour un tour est la suivante : tant que la condition de fin n'est pas réalisée (plus de cartes dans la pioche, ou le joueur qui décide d'aborder avec un bateau) la méthode play() de Player est appelée à chaque tour dans la méthode play() de Boardgame.

5.3.1 Automatisation de l'abordage

Lorsque le joueur possède un bateau et que la pioche de cartes de navigation est vide, le programme propose d'entamer la phase de pose du bateau. Une fois que le joueur a accepté de commencer la phase de pose du bateau, il doit choisir une position (x,y et rotation de la tuile bateau). Si le placement est validé, la phase de récompenses commence.
Il a fallu donc à la fois automatiser le choix d'entamer ou non la pose du bateau, et également automatiser le placement du bateau : i.e. calculer les coups valides possibles.
Pour le bateau à jouer, j'ai implémenté une méthode getValidMoves dans Boardgame qui récupère les coups jouables avec n'importe quelle tuile (dont les bateaux). Le joueur y fait directement appel lorsqu'il lance la phase pour aborder.

5.3.2 Automatisation de l'exploration

Si la phase de pose de bateau n'est pas possible (pioche non vide ou absence de bateau) ou si le joueur choisit de la passer, il peut alors poser une tuile d'exploration.

Dans le cas du joueur automatisé Alexis, celui-ci sélectionne une tuile parmi les trois disponibles, guidé par la carte expédition qu'il vient de piocher. Conformément aux règles, les Alexis placent leur tuile en priorité dans la direction (nord, sud, est ou ouest) par rapport à la dernière tuile posée par l'adversaire, et ce sur la case la plus proche possible. Pour permettre cela, j'ai mis en place des méthodes spécifiques pour chaque direction (*getValidMovesNord()*, *getValidMovesSud()*, etc.), ainsi qu'une méthode générique *getValidMovesDirection()* qui sélectionne la direction à évaluer.

Chaque méthode directionnelle teste l'ensemble des rotations de la tuile choisie et vérifie les conditions de validité du placement (voisinage, absence de tuile déjà posée, conformité avec les règles). Une méthode *isValidPlacement* encapsule cette logique.

Lorsqu'un coup est joué, la position est mémorisée dans la classe Boardgame, ce qui permet d'orienter la recherche de coups lors du tour suivant.

Lors de certaines phases, notamment le début des parties, des tuiles doivent être placées depuis des méthodes de la classe Boardgame. À ce moment, la méthode fait donc également appel aux méthodes pour récupérer les coups possibles.

6 Développement de la grammaire

Les individus sont représentés sous forme de mots acceptés par une grammaire. Cette grammaire a été conçue pour pouvoir représenter tous les Alexis. On peut la définir de cette manière :

mot := "[" liste "]" "B" nombre "M" ['-'] nombre
liste := type chiffre ressource chiffre chiffre chiffre ("," type chiffre ressource chiffre chiffre chiffre)*
type := 'E' | 'A'
ressource := "FR" | "FL" | "LE"
nombre := chiffre (chiffre)*
chiffre := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

Voici un exemple de mot accepté : [E1FR123,E2FL230,E0LE103,E0FR230,E1LE103,A1FR123,A2FL230]B55M1

Il y a trois segments distincts : une liste des cartes d'expédition (**[E1FR123,E2FL230,E0LE103,E0FR230,E1LE103, A1FR123,A2FL230]**), le nombre de points de base (**B55**), et le nombre de points par maison placée (**M1**).
Une carte d'expédition est composée de quatre informations : le type de la carte (aborder ou explorer, 'A' ou 'E'), la carte à choisir dans la pioche (0, 1 ou 2), le type de fruit qui constitue la mission (fruit, flower ou leaf, "FR", "FL", ou "LE"), et l'ordre des directions (0, 1, 2 ou 3). Pour l'encodage des directions : 0 = nord, 1 = est, 2 = sud, 3 = ouest. Voici le schéma provenant des règles du jeu qui l'explique :

Figure 6: Illustration et explications d'une carte expédition

Chaque carte d'expédition peut être vue comme un petit automate fini déterministe (AFD) à trois états. Si aucune tuile ne peut être placée dans la première direction indiquée, l'automate passe à l'état suivant pour tester la deuxième direction, puis la troisième en cas d'échec.

Et voici son implémentation :

```
class Expedition
{
    public:
        Expedition(Action action, std::vector<Direction> directions, int card,
            MissionCard mission); // constructeur
        virtual Expedition* clone() const; // méthode clone
        int getCard() const; // getter pour _card
        std::vector<Direction> getDirections() const; // getter pour _directions
        Action getAction() const; // getter pour _action
        MissionCard getMission() const; // getter pour _mission
    private:
        Action _action; // action à jouer (explorer ou aborder)
        std::vector<Direction> _directions; // ordre des directions à jouer
        int _card; // indice de la carte à jouer dans la pioche
        MissionCard _mission; // mission à remplir pour placer une maison
};
```

J'ai donc implémenté un analyseur syntaxique qui traite cette grammaire, et j'ai ajouté une structure Individual qui contient les informations d'un mot (mot accepté ou non : accepted, liste de cartes : list, nombre de points de base : numberB, nombre de points par maison : numberM), le parser renvoie un type Individual. Ensuite, il ne reste plus qu'à créer un Alexis avec ces informations.

```
struct Individual {
    bool accepted;
    std::vector<std::string> list;
    int numberB; // nombre après B
    int numberM; // nombre après M
};
```

```
Individual individu1 = parse(input1);
Individual individu2 = parse(input2);
players.push_back(std::make_unique<Alexis>(ressource::player1, *this,
    AlexisCard(individu1.numberB,individu1.numberM), individu1.list));
players.push_back(player_number >= 2 ?
    std::make_unique<Alexis>(ressource::player2, *this,
    AlexisCard(individu2.numberB,individu2.numberM), individu2.list) :
    std::make_unique<Alexis>(ressource::player2, *this,
    AlexisCard(individu2.numberB,individu2.numberM), individu2.list));
```

Alexis doit également être capable de générer sa main de cartes d'expédition à partir de la liste. Une méthode *generateExpeditionHand()* transforme les chaînes de caractères de la liste fournie par l'individu en un ensemble de cartes d'expédition, stockées dans un vecteur. Après cela, il est possible d'exécuter une partie à partir de deux individus générés par la grammaire. En voici l'implémentation d'un test (dans le main) :

```
double score1 = 0;
double score2 = 0;
double nbP = 0;
double nombreDeParties = 100;
while(nbP < nombreDeParties)
{
    Boardgame
        board("[E1FR123,E2FL230,E0LE103,E2FR123,E0FR230,E1LE103,A1FR123,A2FL230,
        A0LE103,A2FR123,A0FR230,A1LE103]B55M1",
        "[E1FR123,E2FL230,E0LE103,E2FR123,E0FR230,E1LE103,A1FR123,A2FL230,
        A0LE103,A2FR123,A0FR230,A1LE103]B55M1");
    std::pair<int,int> scores(board.play());
    score1 = score1 + scores.first;
    score2 = score2 + scores.second;
    nbP++;
}
double ratio1 = 0;
ratio1 = score1/nbP;
std::cout << "Score moyen du joueur 1 : " << ratio1 << std::endl;

```

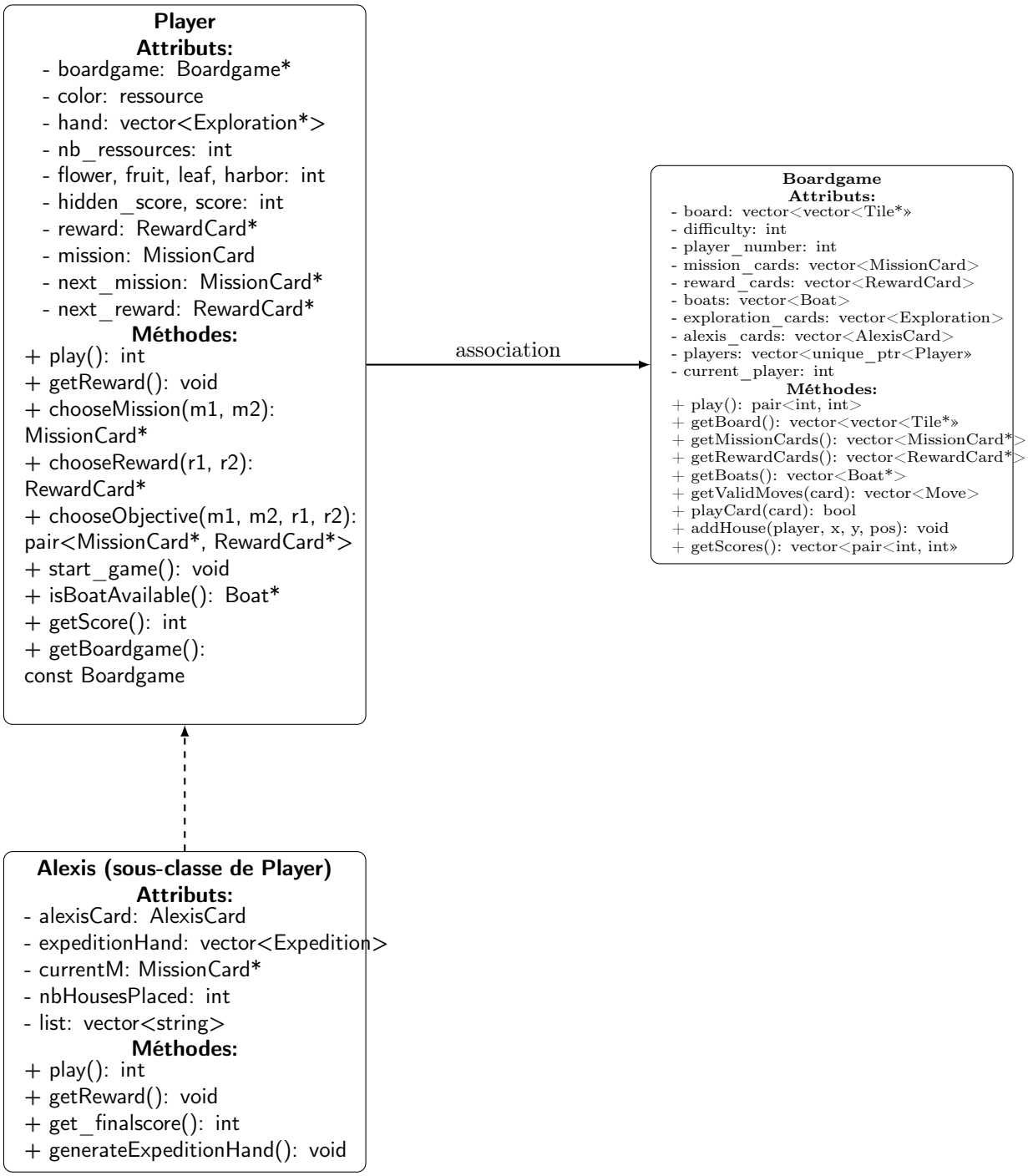
```
17 double ratio2 = 0;
18 ratio2 = score2/nbP;
19 std::cout << "Score moyen du joueur 2 : " << ratio2;
```

Les deux individus identiques suivants vont donc s'affronter :
individu 1 = [E1FR123,E2FL230,E0LE103,E2FR123,E0FR230,E1LE103,A1FR123,A2FL230,A0LE103,A2FR123,A0FR230,A1LE103]B55M1
individu 2 = [E1FR123,E2FL230,E0LE103,E2FR123,E0FR230,E1LE103,A1FR123,A2FL230,A0LE103,A2FR123,A0FR230,A1LE103]B55M1

On obtient ce résultat suite à l'exécution :

```
Score moyen du joueur 1 : 59.53
Score moyen du joueur 2 : 58.77
```

On remarque d'ailleurs que bien que les deux individus soient les mêmes, ils ont des scores différents. L'individu est meilleur en tant joueur 1.
Voici le diagramme de classe des classes Player, Alexis, et Boardgame :



7 Développement de la programmation génétique

Maintenant qu'il est possible de jouer des parties à partir d'une grammaire, il reste à implémenter la programmation génétique et ses différents algorithmes.

7.1 Génération des individus

Tout d'abord il faut générer la population initiale, pour cela j'ai implémenté une méthode *generateRandomIndividual()*. Pour commencer, il faut générer des cartes d'expédition.
On génère d'abord un type de carte (aborder ou explorer), puis l'indice de la carte à choisir, le type de la ressource (c'est une contrainte qui sert pour marquer des points), et enfin un ordre pour les directions (les différents ordres possibles suivent la logique des règles des Alexis).

Une fois ces étapes réalisées on obtient une carte expédition. On répète cela un nombre de fois aléatoire pour obtenir une liste complète. Après cela il suffit de générer deux nombres : le nombre de points de base, et le nombre de points par maison. L'implémentation de cette logique :

```
int E = digitDist(gen);
int A = digitDist2(gen);
std::string individual = "[";
individual += generateRandomList(E,A);
individual += "]";

individual += "B";
individual += std::to_string(pointsBDist(gen));

individual += "M";
individual += std::to_string(pointsMDist(gen));
```

La chaîne de caractères qui représente un individu aléatoire peut maintenant être générée !

7.2 Sélection des individus

Pour sélectionner les individus, j'ai fait le choix d'utiliser la méthode par tournoi.
L'ensemble des individus vont s'affronter entre eux (5 parties sont jouées pour chaque affrontement), si un individu gagne il obtient 1 point, s'il y a une égalité les deux individus obtiennent 0.5 points, et si l'individu perd il obtient 0 point. À partir de cela, un classement par points est obtenu, et on peut donc conserver uniquement la première moitié.

7.3 Croisement des individus

Pour le croisement, le meilleur (pour garder les gênes qui excellent) et le pire individu (pour garder de la diversité génétique) sont toujours gardés. Ensuite tous les individus se croisent pour former le reste de la nouvelle génération.
Il pourrait être intéressant de faire en sorte que les meilleurs aient plus d'enfants que les moins bons, mais cela n'a pas été implémenté.

7.4 Mutation des individus

En termes de mutation, un taux de mutation est défini, un nombre aléatoire est généré et s'il est en dessous de ce taux alors l'individu subit une mutation. Un des trois champs est sélectionné (liste des cartes expédition, points de base, points par maison placée) puis une valeur (ou liste) aléatoire remplace le champ actuel, en suivant le même mécanisme de génération que pour la population initiale.
Une amélioration serait de permettre de ne muter qu'un seul élément de la liste de cartes, comme dans la méthode de mutation "Node Replacement" pour permettre d'affiner les mutations.

8 Résulats

Voici les résultats après l'exécution du programmme sur 150 générations, avec une population de taille 100, et un taux de mutation de 0.25.

Génération	Meilleur individu	Fitness
1	[E2FR012,A1FL032,A2FR012,E0FL103,E0FL032,E1FR301,A1FR032,A0FR032, E0LE301,A0FR210,E1FR012,E0FR230,E0FR301,E2LE032]B72M1	75.866
10	[E2LE012,E2FL103,E2LE210,A0LE230,A1LE210,E0FR123,A2FR301,A1LE123, E2FR103,E0FR103,E1FL103]B76M1	79.842
20	[E0LE210,E2FL103,E2LE210,A0LE230,A1LE210,E2LE032,A2FR301,A1LE123, E2FR103,E0FR103,E1FL103]B76M1	79.557
30	[E0LE210,E2FL103,E2LE210,A0LE230,A1LE210,E2LE032,A2FR301,A1LE123, E2FR103,E0FR103,E1FL103]B76M1	79.624
40	[E0LE210,E2FL103,E2LE210,A0LE230,A1LE210,E2LE032,A2FR301,A1LE123, E2FR103,E0FR103,E1FL103]B76M1	79.594
50	[E0LE301,E2FL103,E2LE210,A0LE230,A1LE210,E2LE032,A2FR301,A1LE123, E2FR103,E0FR103,E1FL103]B76M1	79.651
60	[A0FL032,E1LE103,E0LE032,E1FR210,A1LE123,E2LE032,A2FR301,A1LE123, E2FR103,E0FR103,E1FL103]B76M1	80.303
70	[E0LE301,E2FL103,E2LE210,A0LE230,A1LE210,E2LE032,A2FR301,A1LE123, E2FR103,E0FR103,E1FL103]B76M1	79.710
80	[E0LE301,E2FL103,E2LE210,A0LE230,A1LE210,E2LE032,A2FR301,A1LE123, E2FR103,E0FR103,E1FL103]B76M1	79.692
90	[E0FL012,E0LE301,E0LE123,E1LE103,E2LE123,A2FL321,A2FR210,A2FL012, A2FR012,A1FR012,A0FL032]B76M1	79.837
100	[E0LE301,E2FL032,E2LE210,A0LE230,A1LE210,E2LE032,A2FR301,A1LE123, E2FR103,E0FR103,E1FL103]B76M1	79.656
110	[E0LE301,E2FL032,E2LE210,A0LE230,A1LE210,E2LE032,A2FR301,A1LE123, E2FR103,E0FR103,E1FL103]B76M1	79.735
120	[E0LE301,E2FL032,E2LE210,A0LE230,A1LE210,E2LE032,A2FR301,A1LE123, E2FR103,E0FR103,E1FL103]B76M1	79.762
130	[E0LE301,E2FL032,E2LE210,A0LE230,E0LE230,E0FR230,E2FR321,A1LE123, E2FR103,E0FR103,E1FL103]B76M1	80.001
140	[E0LE301,E2FL032,E2LE210,A0LE230,E0LE230,E0FR230,E2FR321,A1LE123, E2FR103,E0FR103,E1FL103]B76M1	79.985
150	[E0LE301,E2FL032,E2LE210,A0LE230,E0LE230,E0FR230,E2FR321,A1LE123, E2FR103,E0FR103,E1FL103]B76M1	80.033

Table 1: Meilleurs individus et leur fitness pour chaque génération (1e série)

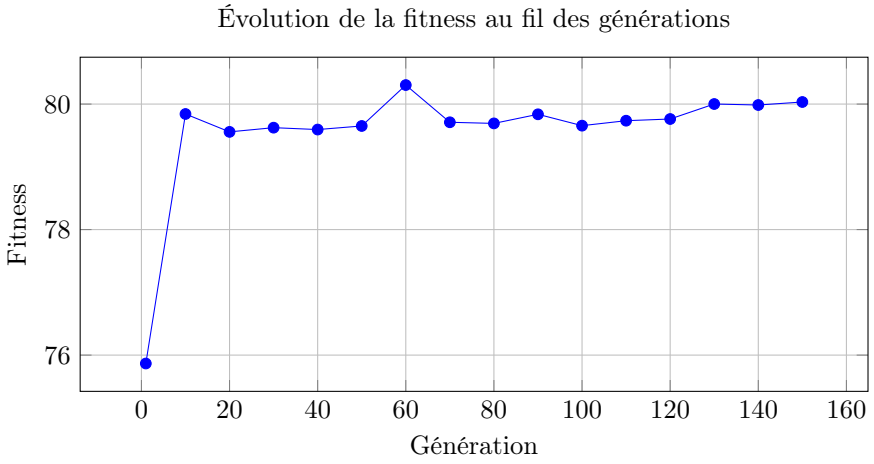


Figure 7: Courbe de la fitness du meilleur individu par génération (1e série)

On remarque que la fitness (score moyen de 1000 parties contre un individu fixé) stagne rapidement autour de 79,5-80,3.
Pour cette prochaine exécution, les 6 alexis ont été ajoutés en début de partie à la population initiale. Avec toujours une population de 100 individus, un taux de mutation de 0.25, et 150 générations.

Voici le classement de la population initiale suite à un tournoi (il y a une flèche à côté des individus qui sont des Alexis pour indiquer leur niveau) :

```
individu 1 : [A1FR123,A1FR230,A0FL123,A0FL012,A2LE230,A2LE301,E0FL123,E2LE012,E2LE301,
E2LE123,E1FR230,E1FR012]B76M0 <- ALEXIS 6
...
E0FL301]B50M5 <- ALEXIS 3
...
individu 12 : [A2LE301,A0FL012,A2LE230,E1FR230,A1FR230,A1FR123,A0FL123]B75M-5 <- ALEXIS 4
...
individu 22 : [A1FR123,E0FL123,A2LE230,A2LE301,E2LE012,E2LE301,A1FR230,E2LE123,A0FL123,
A0FL012,E1FR230,E1FR012]B55M1 <- ALEXIS 5
...
individu 34 : [E0FL230,E0FL123,A1FR123,E2LE012,E2LE301,A1FR230,E2LE123,E1FR230,
E1FR012,E0FL301,A2LE230,A0FL123]B45M1 <- ALEXIS 2
...
individu 38 : [A2LE301,E1FR301,A0FL012,A2LE230,E0FL230,E0FL123,E2LE012,
E2LE301,E2LE123,E1FR230,E1FR012,E0FL301]B40M1 <- ALEXIS 1
```

De manière étonnante, Alexis 3 performe mieux que Alexis 4 et 5. Également Alexis 4 performe mieux que Alexis 5. Par contre Alexis 6 est bien premier à la première génération.
Ce qui peut expliquer les surperformances de Alexis 3 sont qu'il est capable de gagner 5 points par maison placée (contre 1 pour Alexis 5, et Alexis 4 qui perd 5 points).
Pour expliquer les meilleures performances de Alexis 4 comparé à Alexis 5, l'intuition est de penser que sa très large supériorité en points de base (75 contre 50) lui permet de garder de l'avance dans beaucoup de parties.
Également, il y a différentes chaînes de caractères qui peuvent représenter un même Alexis (car les cartes d'expédition sont choisies aléatoirement dans le jeu original, différentes variantes du même Alexis sont possibles), donc peut-être que ces versions des Alexis avec ces cartes d'expédition précisémment surperforment ou sous-performent par rapport à leur moyenne.
Mais évidemment tous ces éléments ne sont que des hypothèses, et il faudrait tester plusieurs fois de plus de suite pour vérifier que ce classement est représentatif.
Le programme arrive à générer dès l'initialisation des individus qui battent Alexis 1 à 5. Mais il y a également des individus qui sont moins bons que les Alexis 1 à 5.

Voici les résultats de ces individus sur 150 générations :

Génération	Meilleur individu	Fitness
1	[E0FR301,E2FL230,A2FR032,A0FL012,A2LE230,A2LE301,E0FL123,E2LE012, E2LE301,E2LE123,E1FR230,E1FR012]B76M0	76.858
10	[E0LE301,E2FL032,E2FL032,E1LE321,A1LE123,E2LE210,E2LE210,E1FL012] B76M1	80.225
20	[E0LE301,E2FL032,E2FL032,E1LE321,A1LE123,E2LE210,E2LE210,E1FL012] B76M1	80.258
30	[E1FL103,E2FR032,E0FL032,E1LE321,A1LE123,E2LE210,E2LE210,E1FL012] B76M1	80.377
40	[E1FL103,E2FR032,E2FL032,E1LE321,A1LE123,E2LE210,E2LE210,E1FL012] B76M1	80.348
50	[E1FL103,E2FR032,E2FL032,E1LE321,A1LE123,E2LE210,E1FL210,E2LE012, E0LE012,E2FR301]B76M1	80.383
60	[E1FL103,E2FR032,E2FL032,E1LE321,A1LE123,E2LE210,E1FL210,E2LE012, E0LE012,E2FR301]B76M1	80.291
70	[E1FL103,E2FR032,E2FL032,E1LE321,A1LE123,E2LE210,E1FL210,E2LE012, E0LE012,E2FR301]B76M1	80.330
80	[E1FL103,E2FR032,E2FL032,E1LE321,A1LE123,E2LE210,E1FL210,E2LE012, E0LE012,E2FR301]B76M1	80.245
100	[E1FL103,E2FR032,E2FL032,E1LE321,A1LE123,E2LE210,E1FL230,E0LE012, E1FR321,E2FR301]B76M1	80.281
120	[E1FL103,E2FR032,E2FL032,E1LE321,A1LE123,E2LE210,E1FL210,E2LE012, E0LE012,E2FR301]B76M1	80.336
140	[E1FL103,E1FR301,E2FL032,E1LE321,A1LE123,E2LE210,E1FL210,E2LE012, E0LE012,E2FR301]B76M1	80.303
150	[E1FL103,E1FR301,E2FL032,E1LE321,A1LE123,E2LE210,E1FL210,E2LE012, E0LE012,E2FR301]B76M1	80.372

Table 2: Meilleurs individus et leur fitness pour chaque génération (2e série)

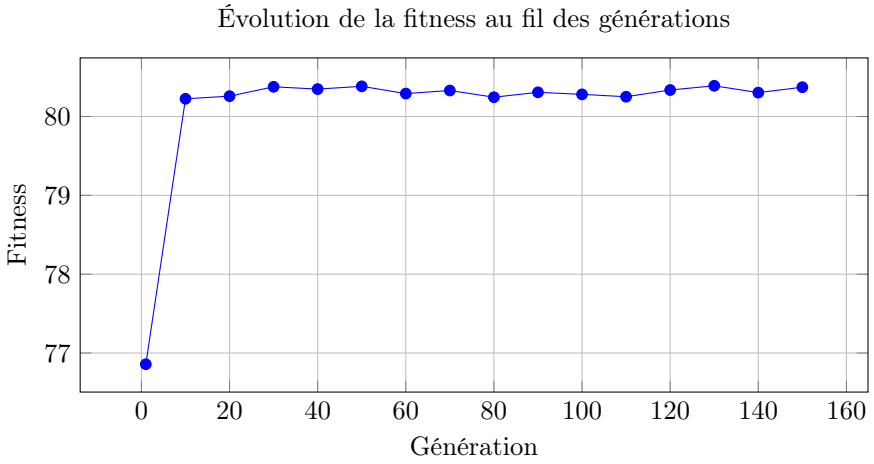


Figure 8: Courbe de la fitness du meilleur individu par génération (2e série)

Dès la dixième génération ce n'est plus Alexis 6 qui est le meilleur. Ensuite à partir de la vingtième génération c'est un individu qui a un nombre de points par maison différent. Rajouter les Alexis n'a donc pas spécialement améliorer les performances sur 150 générations.
Cette fois-ci les meilleurs individus stagnent entre 80,2 et 80,38. On remarque que le meilleur individu sur les 70 dernières générations est toujours le même.

Nouveau test : cette fois l'exécution a été faite sur 250 générations, avec une population de 100 individus, et un taux de mutation de 0.25. Les Alexis étaient également introduits dans la population initiale.

Voici les résultats :

Génération	**Meilleur individu**	**Fitness**
1	[E2LE301,A0FR210,A1FR012,E1LE301,E0FL012,A1FL032, E1LE012,E0FR012,E1FL210,E2LE230,A0LE123,E2FR123, A2LE032,E1FR301,A2FL103]B76M1	80.257
10	[A1FL210,A2LE032,A2LE012,E1FL012,A2LE210,A0LE103, E2FL012,E0LE123,E0FR103,E1LE321,E0FR012,E1FL032, E1FR032,E1FL301]B76M1	80.118
20	[A1FL210,A2LE032,A2LE012,E1FL012,A2LE210,A0LE103, E2FL012,E0LE123,E0FR103,E1LE321,E0FR012,E1FL032, E1FR032,E1FL301]B76M1	80.066
30	[A1FL210,A2LE032,A2LE012,E1FL012,A2LE210,A0LE103, E2FL012,E0LE123,E0FR103,E1LE321,E0FR012,E1FL032, E1FR032,E1FL301]B76M1	80.114
40	[A1FL210,A2LE032,A2LE012,E1FL012,A2LE210,A0LE103, E2FL012,E0LE123,E0FR103,E1LE321,E0FR012,E1FL032, E1FR032,E1FL301]B76M1	80.077
50	[A1FL210,A2LE032,A2LE012,E1FL012,A2LE210,A0LE103, E2FL012,E0LE123,E0FR103,E1LE321,E0FR012,E1FL032, E1FR032,E1FL301]B76M1	80.155
60	[A1FL210,A2LE032,A2LE012,E1FL012,A2LE210,A0LE103, E2FL012,E0LE123,E0FR103,E1LE321,E0FR012,E1FL032, E1FR032,E1FL301]B76M1	80.070
70	[A1FL210,A2LE032,A2LE012,E1FL012,A2LE210,A0LE103, E2FL012,E0LE123,E0FR103,E1LE321,E0FR012,E1FL032, E1FR032,E1FL301]B76M1	80.047
80	[A1FL210,A2LE032,A2LE012,E1FL012,A2LE210,A0LE103, E2FL012,E0LE123,E0FR103,E1LE321,E0FR012,E1FL032, E1FR032,E1FR123]B76M1	80.167
90	[A1FL210,A2LE032,A2LE012,E1FL012,A2LE210,A0LE103, E2FL012,E0LE123,E0FR103,E1LE321,E0FR012,E1FL032, E1FR032,E1FR123]B76M1	80.059
100	[A1FL210,A2LE032,A2LE012,E1FL012,A2LE210,A0LE103, E2FL012,E0LE123,E0FR103,E1LE321,E0FR012,E1FL032, E1FR032,E1FR123]B76M1	80.139
150	[E1FR210,A2LE032,A2LE012,E1FL012,A2LE210,A0LE103, E2FL012,E0LE123,E0FR103,E1LE321,E0FR012,E1FL032, E1FR032,E1FR123]B76M1	80.188
200	[E1FR210,A2LE032,A2LE012,E1FL012,A2LE210,A0LE103, E2FL012,E0LE123,E0FR103,E1LE321,E0FR012,E1FL032, E1FR032,E1FR123]B76M1	80.240
250	[E1FR210,A2LE032,A2LE012,E1FL012,A2LE210,A0LE103, E2FL012,E0LE123,E0FR103,E1FL012,E0FR012,E1FL032, E1FR032,E1FR123]B76M1	80.374

Table 3: Meilleurs individus et leur fitness pour chaque génération (3e série)

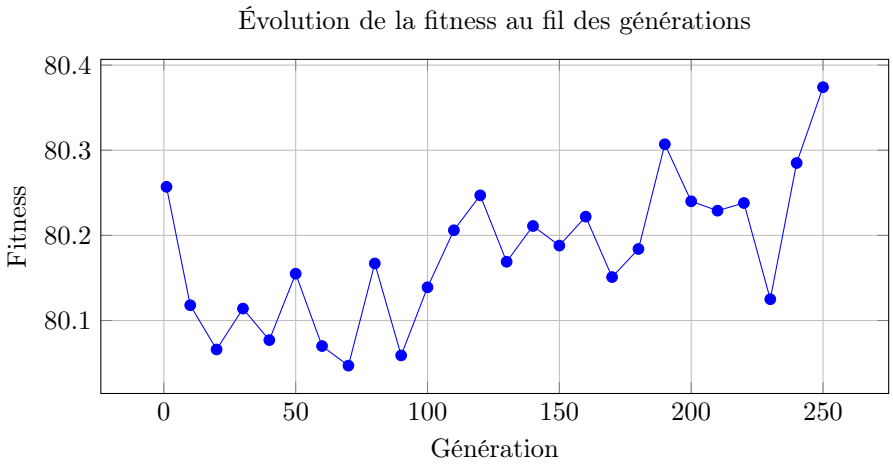


Figure 9: Courbe de la fitness du meilleur individu par génération (3e série)

On remarque que malgré les générations supplémentaires les individus oscillent entre 80 et 80,4, il y a une légère perte de stabilité mais qui serait à vérifier sur le long terme.

9 Conclusion

En conclusion, le programme final propose des résultats satisfaisants en termes de programmation génétique et permet de représenter tous les Alexis, et même si certains points peuvent être améliorés, cela reste un projet complet, fonctionnel, et qui répond au problème initial.

Ce stage a été pour moi l'occasion d'à la fois découvrir la programmation génétique, de continuer à apprendre à s'adapter à une implémentation existante, et d'être capable de gérer son temps pour pouvoir proposer des solutions satisfaisantes à des problèmes en temps voulu. Et tout cela en approfondissant ma découverte du monde de la recherche.

Les difficultés principales rencontrées ont été sur les débuts avec l'automatisation du jeu, et à la fin avec la programmation génétique où il me restait très peu de temps pour l'implémenter. Écrire une grammaire à la fois capable de décrire tous les Alexis, et de proposer une certaine liberté pour proposer de nouvelles intelligences artificielles, a également pu être un point complexe.

Quelques pistes d'améliorations qui pourraient être intéressantes : modifier certaines structures pour réduire le temps de calcul, générer des individus moins proches des Alexis pour permettre des intelligences artificielles différentes, essayer d'autres méthodes de sélection, de croisement et de mutation.

10 Sommaire des schémas

List of Figures

List of Tables

11 Annexes

https://omnilogie.fr/images/O/0e6e1fcc31f8281e7d1445bd64714aa2.jpg : source pour l'image de labyrinthe utilisée durant l'étude bibliographique de la programmation génétique.

https://cdn.1j1ju.com/medias/7d/12/40-small-islands-regle.pdf : règles du jeu de société **Small Islands**

12 Sources et bibliographie

Poli, Riccardo & Langdon, William & Mcphee, Nicholas. (2008). A Field Guide to Genetic Programming.

McConaghy, Trent & Vladislavleva, Ekaterina (Katya) & Riolo, Rick. Genetic Programming Theory and Practice 2010: An Introduction.

Koza, J.R. (1992), Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press

Nordin, Peter & Keller, Robert & Francone, Frank. (1998). Genetic Programming: An Introduction on the Automatic Evolution of computer programs and its Applications.

Holland, J. H. (1975). Adaptation in Natural and Artificial Systems. University of Michigan Press.