INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

Louis Bagot
Edouard Donzé
Simon Delecourt

Rapport de projet semestriel

"Flappy Whale" IA

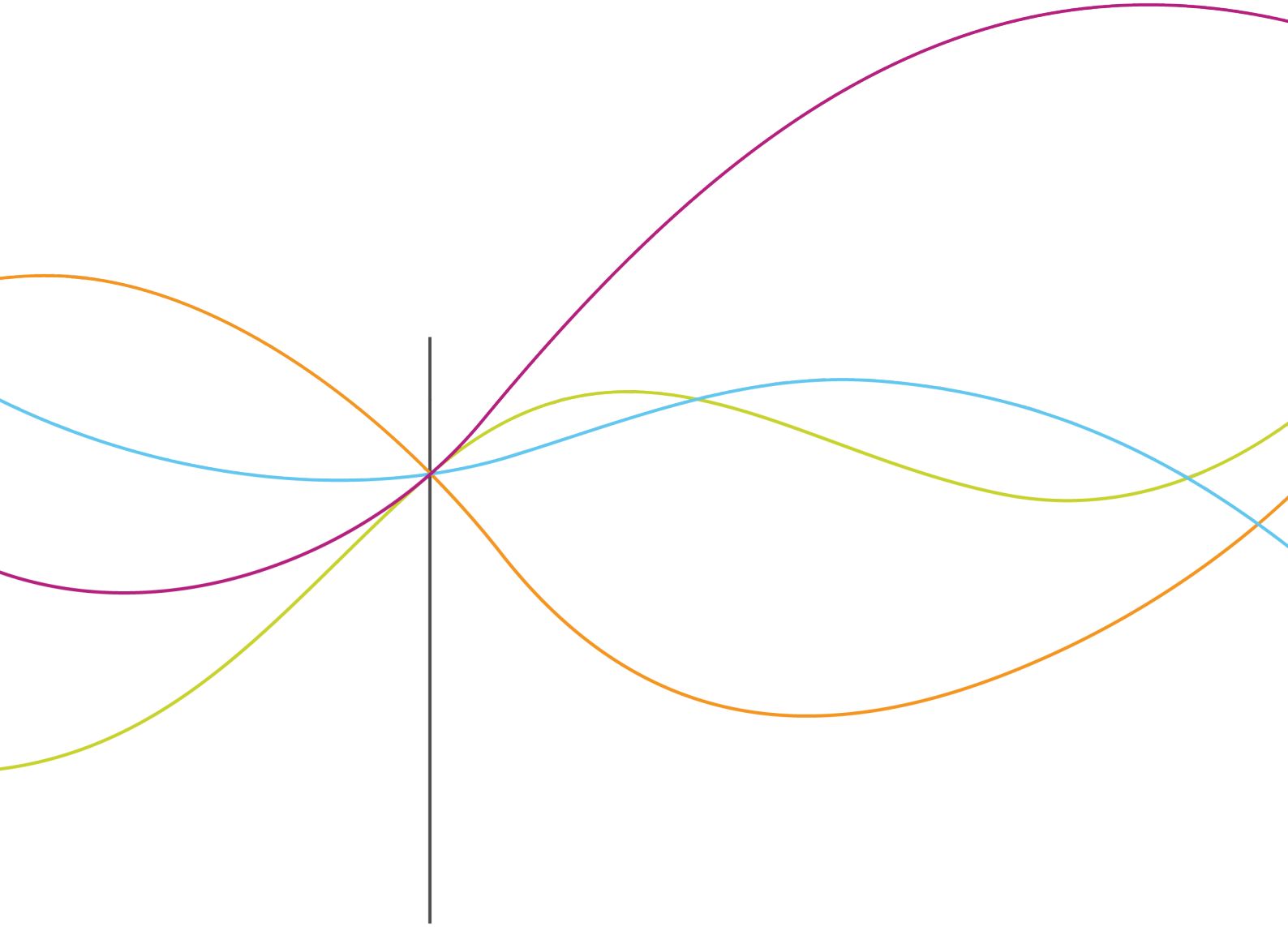


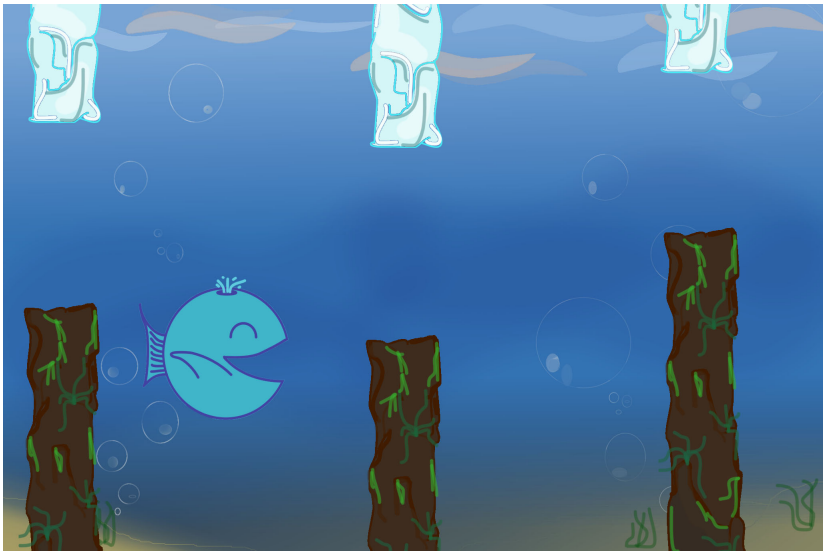
Table des matières

1 Introduction

L'objectif de ce projet est d'appliquer des méthodes de Machine Learning sur un jeu, originellement une application smartphone : "Flappy Bird". Nous étions par hasard tombé sur une vidéo sur le sujet, et avons choisi de le traiter, avec ce jeu en particulier, car il est relativement simple et a l'avantage de fournir un environnement aléatoire, ce qui est intéressant pour tester un algorithme qui se veut intelligent - en effet cela permet de contrer la simple mémorisation de l'environnement.

Le jeu consiste à contrôler un oiseau afin qu'il passe une série d'obstacles. Il n'est possible d'effectuer qu'une action, qui ordonne à l'oiseau de s'élever. L'autre ordre implicite est donc de le laisser redescender. Si l'oiseau touche un obstacle ou qu'il atteint le plafond ou le sol, il "meurt" et sa partie se termine.

Nous avons donc implémenté ce jeu en Java en prenant la liberté de changer l'univers (sans impact sur les règles ou la physique) : le joueur contrôle à présent une baleine ronde sous l'océan, d'où le nom "Flappy Whale".



Les différents concepts que nous avons abordés durant ce projet sont en premier lieu la conception orientée objet, les algorithmes génétiques et les réseaux de neurones.

2 Principe général des concepts utilisés

2.1 Les Algorithmes Génétiques

2.1.1 Concepts d'Apprentissage par Renforcement

Les algorithmes génétiques, dans le cadre de notre approche, appartiennent à la famille des algorithmes de "Machine Learning" (*Apprentissage Automatique* en français) qui ont pour objectif d'améliorer la performance de l'algorithme par l'apprentissage. Plus précisément, ils font partie de la plus petite famille des algorithmes de "Reinforced Learning" (*Apprentissage par Renforcement*). Ces derniers cherchent à améliorer les performances de l'agent, plongé dans un environnement, à partir de sa seule expérience. Cela s'oppose par exemple au "Supervised Learning" (*Apprentissage Supervisé*), dans lequel le superviseur humain donne les bonnes réponses (absolues) à l'algorithme, qui apprend à partir de celles-ci.

Une fois l'*environnement* défini, à chaque étape, un agent (contrôlé par l'algorithme par renforcement) va devoir choisir l'*action* à effectuer par rapport à l'*état* dans lequel il se trouve. L'environnement doit pouvoir fournir une *récompense* à l'agent, qui va lui permettre de faire le tri parmi ses comportements.

En théorie, ces algorithmes peuvent atteindre, si l'environnement le permet, une convergence vers un agent au comportement idéal. Ils sont particulièrement efficaces pour les jeux (où les règles - c'est-à-dire l'environnement - sont parfaitement définies), où ils se sont révélés bien plus efficaces que leurs analogues de Supervised Learning, ayant appris à partir des connaissances humaines. Ils permettent donc d'atteindre des paliers inaccessibles aux êtres humains.

2.1.2 Principe

Les algorithmes génétiques s'inspirent de la théorie de *l'évolution Darwinnienne*. Celle-ci stipule que les individus les plus *adaptés* à leur environnement auront le plus de chances de se reproduire, et ainsi de passer leur *gènes* à la génération suivante. Cependant, la passation de gènes est susceptible d'être "altérée" par des *mutations*, qui vont très légèrement modifier l'enfant (en bien ou en mal). Cette dernière idée permet d'assurer la diversité du génôme.

Il est alors possible de traduire ces concepts informatiquement, en identifiant d'abord les idées clé du processus. Tout d'abord, sans perte de généralité, les individus peuvent être séparés en *générations*, où la génération n est engendrée par la génération $n-1$. La génération 0 est alors créée avec des génotypes [1] aléatoires. Chaque individu va être confronté à l'environnement, où il va se voir attribuer un "score" (*fitness*) attestant de sa performance. Suivent ensuite les opérations d'engendrement de la génération suivante, basés sur le modèle Darwinien décrit plus haut :

- La *sélection* : on attribue aux individus aux scores les plus élevés une probabilité plus forte d'être parent.
- Le brassage génétique (*crossover* en anglais) : les gènes de deux parents sont mélangés pour obtenir un nouveau génotype produit de ces deux parents.
- Les *mutations* : l'enfant voit certains de ses gènes modifiés de manière aléatoire.

L'algorithme génétique consiste donc à répéter ces générations successives jusqu'à convergence empirique (performance arbitrairement considérée comme "suffisamment bonne").

2.1.3 Interprétation mathématique

Il est important de préciser certains points mathématiques de cet algorithme, notamment sur la nature et le choix de l'ADN : il sera crutial pour l'existence de solutions. L'approche mathématique sera décrite ici et réutilisée quand nécessaire. Formalisons dans un premier temps l'approche de notre modèle.

Fixons d'abord l'objectif. Nous cherchons à faire agir un agent idéalement dans n'importe quelle situation de notre environnement. Autrement dit, pour S l'ensemble des états de l'environnement et A l'ensemble des actions possibles, nous cherchons la fonction de comportement idéal [2] $f^* : S \longrightarrow A$. Nous noterons $\mathcal{D} = \{f : S \longrightarrow A\}$ l'ensemble de ces fonctions de décision.

1. ensemble de gènes, réduit à l'*ADN* dans notre modèle simplifié
2. Dans notre cas de Flappy Whale, la fonction prend la position de la baleine et des obstacles, et ressort un booléen dictant s'il faut sauter ou non.

Il faut déjà être assuré de l'existence d'une $f^* \in \mathcal{D}$ (au moins "suffisemment performante") - dans notre cas, on peut deviner empiriquement qu'il est possible de jouer parfaitement, et donc que cette fonction existe.

Cette fonction n'a pas nécessairement de forme précise. Nous allons choisir un espace $E \subset \mathcal{D}$ dans lequel nous pensons qu'elle existe[3], et chercher à s'en approcher. C'est ici qu'intervient l'idée de génétique.

Nous allons générer une fonction de décision $f_E \in E$ associée à un ensemble de paramètres, ou gènes. Un ADN particulier correspond alors à un point, ou vecteur, dans l'espace des gènes que l'on notera Θ. La fonction f_E est identique pour toute la population : un vecteur ADN est alors entièrement responsable du comportement de l'individu dans l'environnement ; notons la fonction particulière qu'il engendre $f_{E,ADN}$.

Notre objectif se résume alors à la recherche du point $ADN^* \in \Theta$ tel que $f_{E,ADN^*} = f^*$. Le problème peut alors être réexprimé comme problème d'optimisation où il faut minimiser la distance à f^*, ou maximiser la fitness. En créant la génération 0 de taille N, l'algorithme génétique produit aléatoirement N points $ADN \in \Theta$, puis les opérations de crossover et de mutation vont respectivement rapprocher des points, et explorer leur voisinage. La sélection permet de réaliser ces opérations sur les points les plus intéressants, c'est-à-dire les plus proches des dépressions dans le graphe des distances à ADN^*.
Cependant cette dernière observation soulève un problème qui est majeur chez l'algorithme génétique : s'il existe des points de minimum local sur Θ,[4] l'algorithme n'aura aucun moyen de les éviter et plongera vraisemblablement dessus, surtout si l'espace est vaste et ces minimum locaux fréquents.
Les seuls moyens d'éviter ceci sont d'avoir une génération 0 suffisamment variée, une sélection qui ne donne pas trop de pouvoir aux individus immédiatement les plus efficaces, et une amplitude de mutation qui permet de s'extirper des zones de voisinage du minimum.

Tout ceci, avec d'autres paramètres, fait partie de l'alchimie délicate qu'il faut réaliser avec les *hyperparamètres*, c'est-à-dire les paramètres définis arbitrairement au lancement de l'algorithme. Ceci soulève un des plus gros défauts de l'algorithme : son abondance en hyperparamètres rend la recherche de convergence très complexe, et peut même faire douter de l'efficacité d'un modèle ADN.

2.1.4 Hyperparamètres

Nous en avons déjà cité quelques uns, et nous allons ici lister tous les hyperparamètres de l'algorithme avec lesquels il faudra jouer pour chercher à obtenir la convergence.

— **La taille de la population** N. Celle-ci permet d'assurer une diversité essentielle dans le génome, cependant une trop grande population aura évidemment un impact direct sur la vitesse de l'algorithme, en plus d'entraver certaines fonctions de sélection, comme nous le verrons plus tard.

— **La fonction de fitness**. La méthode d'affectation d'une valeur quantitative a une performance n'est pas toujours simple et il est difficile de savoir ce qui poussera l'algorithme au meilleur résultat.

— **Les paramètres de mutation**, que nous avons appelé dans le code *mutProba* et *mutAmpl*.
Le premier est la probabilité qu'une mutation occure chez l'enfant. Ce paramètre est fixé tout au long de l'algorithme. A la création de chaque gène de l'enfant dans son ADN, sous cette probabilité, une mutation occurre.
Le second est l'amplitude de la mutation : nous avons implémenté nos mutations avec une répartition uniforme, que nous centrons et réduisons par ce paramètre. Nous avons fait en sorte que ce paramètre puisse décroître au cours du temps, ce qui réduit l'amplitude des mutations lorsque l'on se rapproche d'un espace solution[5]. La méthode de décroissance de ce paramètre est elle-même arbitraire et vient avec ses propres hyperparamètres (comme la valeur de départ et la vitesse de décroissance).

— **La fonction de sélection**. Le choix de la méthode de sélection est arbitraire mais impacte grandement les résultats. Il faut aussi souligner que la fonction de sélection peut admettre ses propre hyperparamètres, que nous détaillerons dans la partie correspondante.

3. Cet espace sera, par exemple, l'espace des réseaux de neurones (ensemble des poids et biais d'un réseau quelconque)
4. Ceci peut être dû au fait que certains modèles ADN, bien qu'éloignés d'ADN^*, permettent tout de même d'obtenir une très bonne fitness.
5. Ceci est implicitement défini comme "un voisinage d'un minimum local dans lequel la *fitness* est très forte".

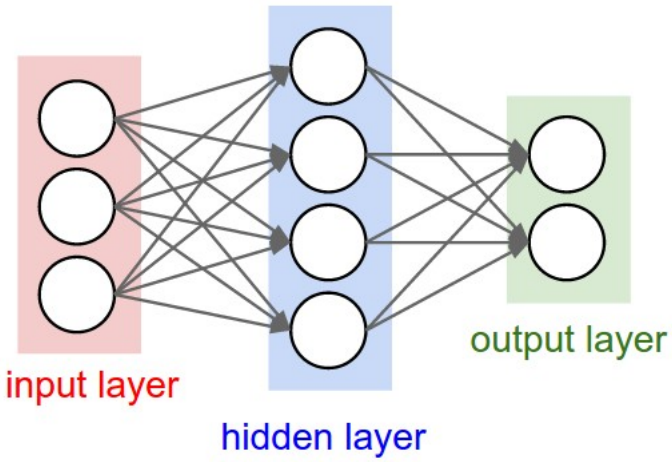
— **Le modèle ADN**. C'est l'hyperparamètre le plus important : il aura le plus grand impact sur les résultats. Il vient aussi avec ses propres hyperparamètres que nous détaillerons.

2.2 Les réseaux de neurones

Un réseau de neurones artificiel est un concept informatique qui s'inspire du modèle du cerveau. Il se schématise comme un graphe dont les sommets sont les *neurones*. Ce graphe est séparé en différentes couches, ou *layers*. Dans le modèle que nous utiliserons, le graphe d'une couche à l'autre est complet.

Mathématiquement, un réseau de neurones est une fonction, avec donc une entrée, une sortie, et ses paramètres. Le vecteur entrée est, schématiquement, la première couche du graphe (*input layer*). De l'autre côté, le vecteur sortie est schématisé par la dernière couche du graphe (*output layer*). Toutes les couches intermédiaires constituent l'ensemble des couches dites "cachées" (*hidden layers*).

Chaque arc du graphe est associé à une valeur réelle nommée *poids* (ou *weight*), qui permettra de calculer la sortie de la fonction à partir d'une entrée donnée. Tous les poids du réseau constituent alors la totalité de ses paramètres, et le but sera d'en trouver les valeurs idéales.
Il est important de préciser que, traditionnellement, à chaque couche est aussi associée un vecteur nommé *biais* ; mais étant donnée la simplicité du problème, nous l'avons ici ignoré (sans conséquences).



L'algorithme permettant d'obtenir la réponse d'une réseau de neurones à partir d'un *input* donné est appelé la *propagation* (parfois aussi *feedforward* dans la littérature). Cette dernière se réalise en calculant une suite d'opérations matricielles qui se résument ainsi [6] :

$$\forall l \in \{2, ..., n\}$$
$$z^l = w^l a^{l-1} + b^l$$
$$a^l = \sigma(z^l)$$

Où n est le nombre de couches du réseau ; w^l la matrice des poids des couches $l-1$ à l ; b^l le vecteur biais sur la couche l, et a^l les activiations, ou valeurs des neurones, de la couche l.
Le vecteur *input* est donné par a^1, et le vecteur *output* est donc donné par a^n. La *fonction d'activation* σ est une application [7] de $\mathbb{R}$ dans $\mathbb{R}$ appliquée élément par élément dans le vecteur z^l.

Il est important de souligner notre motivation quant à l'utilisation de ce modèle pour l'algorithme génétique. Il faut d'abord savoir que ceci aurait pu complexifier la recherche : en effet, un réseau de neurones vient avec une pléthore d'hyperparamètres (nombre de couches cachées et toutes leurs tailles, valeurs initiales des poids et biais...). Cependant dans le cadre d'un problème si simple nous savions que cette partie ne serait pas si complexe.

6. Les exposants dans ces formules sont bien des indices et non des puissances ; les formules sont à appliquer comme un "for" sur la variable muette l sur l'intervalle donnée
7. les applications testées seront décrites plus loin.

De plus nous avons vu que nous cherchons à trouver une fonction particulière. Or, une propriété excessivement importante des réseaux de neurones est que même pour une unique couche cachée, ils constituent un *approximateur universel* de n'importe quelle fonction continue dans un compact de $\mathbb{R}^n$. Cela nous permettait d'être très confiant du fait qu'il existait un réseau de neurones adapté à notre problème. La seule difficulté restante étant d'en trouver l'architecture, qui se révéla particulièrement simple.

3 Application au problème

Nous avons donc cherché à appliquer l'algorithme génétique au jeu de Flappy Whale. Il fallut alors penser a une fonction de fitness, trouver puis implémenter des modèles ADN adaptés, et enfin imaginer puis implémenter différentes méthodes de séléction.

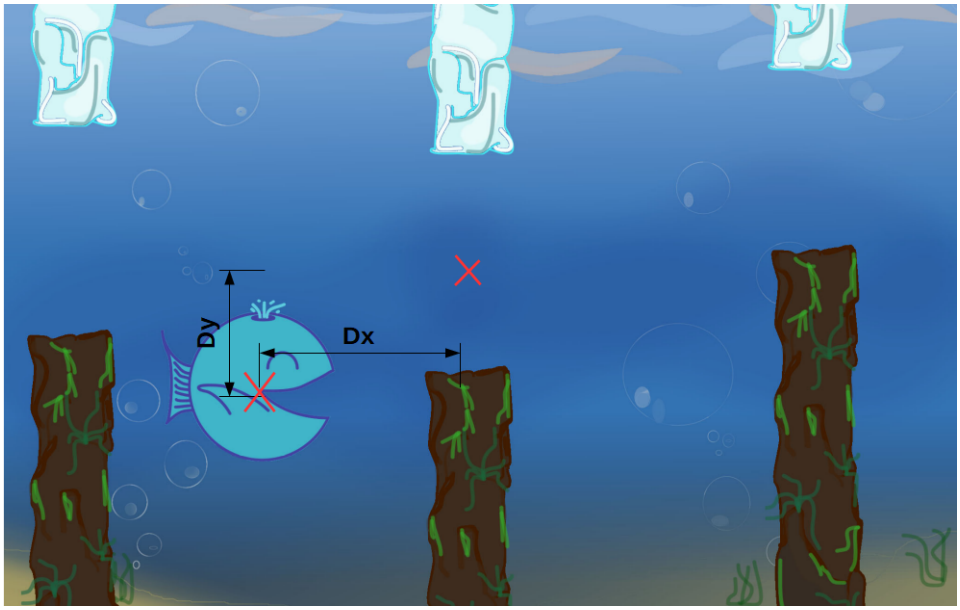
3.1 Fonction de fitness

La fonction de fitness était assez évidente puisque nous cherchions à ce que la baleine aille le plus loin possible ; établir alors $fitness = distanceParcourue_x$ semblait naturel. La vidéo d'origine nous a cependant inspiré pour ajouter un terme de "punition" si la baleine venait s'échouer trop loin du centre de l'obstacle sur l'axe y : $fitness = distanceParcourue_x - distanceCentreObstacle_y$. Cela permet de favoriser les baleines venant s'échouer proche du centre de l'obstacle, face à celles s'échouant à sa base ou trop haut.

3.2 Premier modèle d'ADN : maillage

3.2.1 Implémentation

La première approche à ce problème que nous ayons essayé est assez naïve : elle consiste simplement à mailler les positions relatives de la baleine au centre de l'obstacle. A chaque noeud de ce maillage est alors associé un booléen, qui dictera si cette position relative particulière (Dx, Dy) est propice au saut de la baleine.



Informatiquement, cela se traduit par un tableau de booléens en 2D. Sa version la plus naïve est d'associer à chaque pixel un noeud du maillage. En effet cela permet de déterminer des valeurs intéressantes pour les deux dimensions du tableau.

Nous souhaitons connaître les positions relatives de la baleine, et celle-ci peut se retrouver très haut ou très bas par rapport à l'obstacle. En notant $DIMY$ le nombre de pixels de hauteur de l'écran :

— Si la baleine est très haute par rapport à l'obstacle alors $Dy \approx DIMY$

— Sinon si la baleine est en bas de l'écran et l'obstacle en haut, alors $Dy \approx -DIMY$

On déduit de ceci que le nombre de lignes du tableau de booléens naïf devra être $\approx 2DIMY$. Pour le nombre de colonnes, nous avons simplement estimé que l'ensemble des comportements intéressants se trouvait à proximité de l'obstacle. Nous savons que le jeu ne peut pas générer d'obstacle séparés de moins d'une certaine distance $Obstacle.MINDIST$, nous l'avons donc utilisée comme valeur pour le nombre de colonnes. Si Dx s'avère supérieur à cette valeur, on accède à une colonne réservée qui détermine un comportement par défaut pour ce cas précis.

A partir de ces deux valeurs, nous pouvons simplement décider d'un facteur par lequel les diviser pour effectuer un maillage moins naïf de l'environnement. Un valeur qui nous semblait évidente fut la vitesse des obstacles : en effet ces derniers avancent, à chaque frame [8], d'une distance $Obstacle.SPEED$. Il est donc inutile de mailler tous les pixels puisque beaucoup d'entre eux seront évités. Ce premier maillage permet de grandement réduire la taille du tableau, ce qui était notre principal objectif afin de garantir une convergence rapide des IA. Nous avons alors simplement essayé des multiples de $Obstacle.SPEED$ pour déterminer lequel permettrait de mailler suffisamment tout en gardant un comportement efficace. L'expérience nous a montré qu'un maillage de $4 * Obstacle.SPEED$ était suffisant.

Le modèle ADN associé à ce tableau s'appelle *BoolArray* dans le code. Sa fonction de décision est alors simplement l'accès à la case du tableau correspondant à la position relative à l'obstacle (après réduction sur le maillage).
Pour la génération aléatoire d'un individu, nous avons utilisé une heuristique : nous savons que la baleine doit sauter assez rarement pour rester dans le cadre du jeu. Nous avons alors décidé de ne remplir aléatoirement de TRUE qu'un pourcentage faible du tableau. L'expérience nous a montré que 3% de TRUE fournissait un comportement moyen intéressant pour la génération 0.
Le modèle étant discret, l'implémentation des mutations est assez brutale : en cas de mutation, la valeur val du tableau correspondante prend directement $\mathsf{non}(val)$.

Pour donner une idée de la complexité du travail demandé à l'algorithme, nous pouvons calculer le nombre de paramètres à ajuster :
Le tableau (avec les hyperparamètres donnés jusqu'ici et les paramètres que nous avons choisi pour implémenter le modèle du jeu) fait finalement un taille de 42×17 ce qui fait environ 700 booléens à ajuster. Il faut cependant garder en tête que ce chiffre est dépendant de paramètres arbitraires. Cela signifie que le tableau idéal est 1 parmi $|\Theta_{ba}| \approx 2^{700} \approx 10^{210}$ tableaux possibles [9]. Ce nombre astronomique peut être considérablement réduit en pratique si on considère qu'une partie du tableau est vouée à n'être jamais utilisée (une partie des extrêmités notamment) - cependant même en imaginant qu'une moitié entière du tableau était inutile, nous aurions $2^{350} \approx 10^{105}$ tableaux pratiques différents à explorer. Cela exclut dont toutes tentatives de "force brute" pour remplir le tableau, et justifie l'utilisation de l'algorithme génétique pour explorer Θ_{ba}.

3.2.2 Résultats

Il est important de préciser que dans la partie qui suit, nous allons présenter les résultats pour ce modèle ADN particulier, et donc choisir les autres hyperparamètres de manière empiriquement idéale. Dans le cas du *BoolArray*, il semblerait que ces paramètres soient :
- La taille de la population $N = 1000$
- $mutAtZero = 0.05$, $decrease = true$;
- $selector = RankSelection(0.05)$;

Une étude des hyperparamètres sera faite à la fin de cette section. A chaque présentation des résultats, nous allons exposer deux lancements différents de l'algorithme. En effet celui-ci étant très fortement stochastique, les résultats peuvent énormément varier d'une exécution à l'autre, même avec des hyperparamètres inchangés.

L'algorithme génétique sur modèle ADN des *BoolArray* s'est montré parfaitement fonctionnel. Voici deux exécutions consécutives :

8. Une frame est un tour de boucle du programme principal ; c'est donc le temps minimal de réactivité autorisé au joueur.
9. Avec $|\mathcal{X}|$ le cardinal de l'ensemble $\mathcal{X}$.

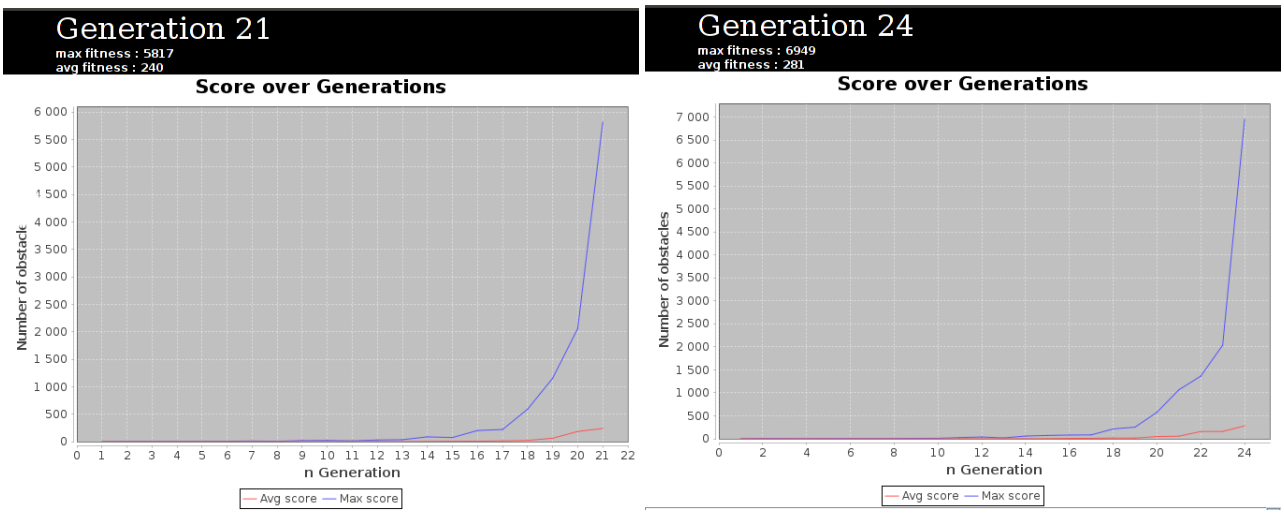


FIGURE 1 – Modèle ADN *BoolArray* converge en une vingtaine de générations, pour $N = 1000$

La courbe bleue correspond à l'évolution du meilleur score au fil des générations tandis que la courbe rouge correspond au score moyen. Le "score" est ici différent de la *fitness*, puisqu'il est égal au nombre d'obstacles traversés - une notion plus intuitive que la fitness pour observer les résultats.

Evidemment, la notion de *convergence* est complexe puisqu'une convergence parfaite correspondrait à une fitness infinie, non vérifiable en pratique. Nous pouvons définir une notion de convergence arbitraire, par exemple par rapport à un temps d'exécution (équivalent à un certain nombre d'obstacles passé). Nous nous sommes arrêtés ci-dessus à 6000 obstacles, que nous pouvons définir comme une mesure arbitraire de vitesse de convergence.

Bien que la courbe trace une exponentielle assez nette, nous pouvons observer ici et sur d'autres exécutions que les courbes peuvent parfois décroître. Ceci est parfaitement normal et est dû à plusieurs facteurs. Tout d'abord, des facteurs d'ordre général (non liés à l'ADN) :

— Un environnement plus coriace. L'environnement est généré aléatoirement et peut parfois demander un plus haut niveau d'aptitude, pour lequel les individus n'ont pas été préparés.
— Les individus en tête dans la génération précédente, qui sont donc principalement responsables des génotypes d'une grande partie de la population, peuvent avoir eu un génotype particulièrement adapté à leur environnement, mais le même ne fonctionnerait pas nécessairement aussi bien sur un autre.
— La sélection donne une certaine chance aux individus les plus faibles de se reproduire ; ceux-ci permettent de garder de la diversité dans le génôme mais peuvent entraver la performance de la génération suivante - surtout s'ils sont nombreux face à l'élite.
— Les mutations sont très arbitraires et généralement assez catégoriques - si la performance d'un individu tenait d'un équilibre fragile enre ses gènes, il est très probable que cet équilibre soit rompu par les mutations à la génération suivante.

Dans ce cas des *BoolArray*, nous pouvons deviner que le tableau 2D agit comme une sorte de mémoire. Les actions à réaliser dans chaque situation sont enregistrées et répétées ; et elles sont ajustées au cours du temps. Toute la population peut alors être performante, mais tomber sur une situation jusqu'alors inconnue, et échouer. Les mutations peuvent même faire oublier à l'individu le comportement à adopter.

Tout ceci n'empêche cependant pas le modèle d'atteindre des records faramineux :

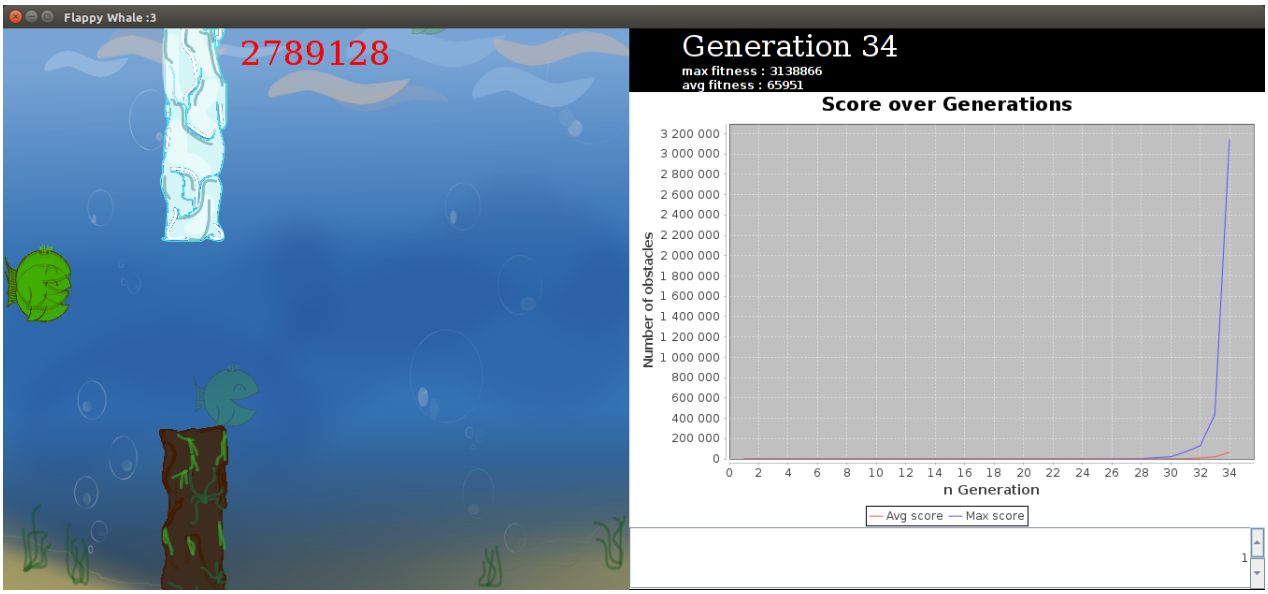


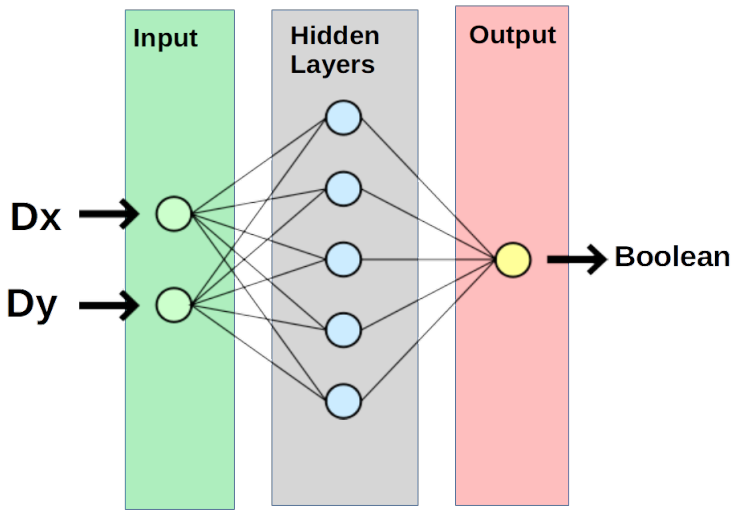
FIGURE 2 – Plusieur *BoolArray* dépassent le million d'obstacles, pour $N = 1000$

3.3 Second modèle ADN : Réseaux de neurones

3.3.1 Implémentation

La seconde approche, comme développée dans la fin de la seconde partie, est d'utiliser un réseau de neurones comme modèle ADN.

Ce réseau aura une *input layer* de deux neurones qui correspondront aux précédents (Dx, Dy), cependant ils sont maintenant transformés linéairement sur $[0, 1]$ afin de garder toutes les valeurs des activations entre 0 et 1 au cours de la propagation. En effet, la *fonction d'activation* que nous choisissons est la fonction sigmoïde, c'est-à-dire $\sigma(x) = \frac{1}{1+e^{-x}}$. Cette fonction est une approximation infiniment dérivable de la fonction de Heaviside (*step function*) ; elle est donc à valeurs dans $]0, 1[$.
La sortie du réseau n'est constituée que d'un unique neurone, et nous choisissons arbitrairement qu'une valeur de sortie $> 0,5$ indiquera l'ordre de sauter.



NB : Le modèle ADN associé au réseau de neurones est simplement nommé *NeuralNet* dans le code.

Comme précisé ultérieurement, nous avons décidé de nous passer de biais pour simplifier le modèle et surtout réduire le nombre de paramètres. En effet comme l'algorithme génétique explore l'espace Θ des paramètres, la réduction de ce dernier permet de simplifier la recherche - cela peut à la fois accélérer l'algorithme et faciliter la convergence, si la réduction a du sens.
Pour l'architecture du réseau, nous nous sommes d'abord inspirés de la vidéo et avons découvert qu'une seule couche cachée de 6 neurones permettait d'obtenir suffisemment de complexité de comportement, et nous avons donc adopté cette valeur. Cela signifie que l'algorithme doit ajuster $2 * 6 + 6 * 1 = 18$ gènes, ou poids du réseau.
Une heuristique habituelle d'initialisation des poids donne qu'il faut les garder assez faibles (pour assurer une généralisation et éviter d'*overfit* [10]). Nous avons empiriquement choisi de générer les poids dans $\left[-\frac{3}{4}, \frac{3}{4}\right]$. Cela signifie que l'espace initial dans lequel nous allons piocher aléatoirement est $\left[-\frac{3}{4}, \frac{3}{4}\right]^{18}$. Cet ensemble est différent de Θ_{nn} (l'ensemble Θ des réseaux de neurones), car les poids peuvent ensuite en sortir. Notons-le $\Theta_{nn,0}$.

3.3.2 Résultats

Encore une fois, nous allons choisir des hyparamètres hors-ADN (mutations, sélection) empiriquement idéaux.

- La taille de la population $N = 1000$
- $mutProba = 0.2$, $mutAtZero = 0.02$, $decrease = true$;
- $selector = rankSelection(0.2)$ - cette valeur peut en fait être encore plus grande;
- 3 neurones dans la couche cachée. [11]

La première chose à remarquer lors du lancement de l'algorithme avec les réseaux de neurones est que l'espace solution a un volume très important. En effet, sur 1000 individus de la génération 0, il n'est pas rare qu'un individu passe la barrière des 100 obstacles. Cela signifie qu'un réseau généré aléatoirement a une chance sur quelques milliers d'être un réseau relativement adapté à résoudre le problème. Autrement dit pour $\mathcal{S}$ l'espace solution, $\exists k \geqslant 1$ relativement petit tel que [12]

$$\mu(\mathcal{S}) = \frac{\mu(\Theta_{nn,0})}{1000k}$$

Pour rappel, nous avions établi que dans le cas des *BoolArray*, la taille de l'espace solution était aux alentours d'un Googolième [13] de la taille de l'espace Θ_{ba}.
Un des exemples les plus extrêmes d'individu très performant en génération 0 est le suivant :

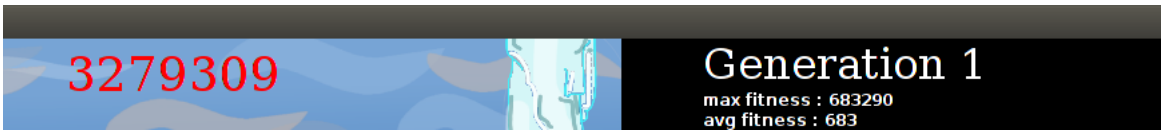


FIGURE 3 – Excellents individus générés alétoirement

Nous sommes tombés, à la génération 0, sur un individu qui a parcouru plus de 650000 obstacles; et la génération 1 donnera naissance à un des individus les plus performants que nous ayons vu, puisque nous ne l'avons pas vu perdre : il nous a fallu arrêter l'algorithme. ($> 20.10^6$ obstacles)

Tout cela prouve que le problème est extrêmement simple à résoudre pour les réseaux de neurones, et on l'observe : la convergence se fait systématiquement en moins de 10 générations. Cela s'explique simplement : la vitesse de convergence est directement liée à la taille relative de $\mu(\mathcal{S})$ à $\mu(\Theta)$, puisque l'algorithme explore Θ à la recherche de $\mathcal{S}$.

10. Overfit : lorsque le modèle se calque trop précisément sur un ensembles d'exemples en particulier, et perd tout pouvoir de généralisation.
11. Nous avons très souvent utilisé 6 neurones dans nos expérimentations, mais il s'est avéré qu'avec 3 les résultats étaient encore pertinents
12. On note ici $\mu(\mathcal{X})$ la mesure (ou volume) de l'ensemble $\mathcal{X}$
13. $1Googol = 10^{100}$

Il faut noter que, les *NeuralNet* étant légèrement plus lents à répondre, il est un peu plus difficile d'obtenir des résultats pour un grand nombre de générations.
Voici finalement les résultats consécutifs obtenus, pour $N = 1000$:

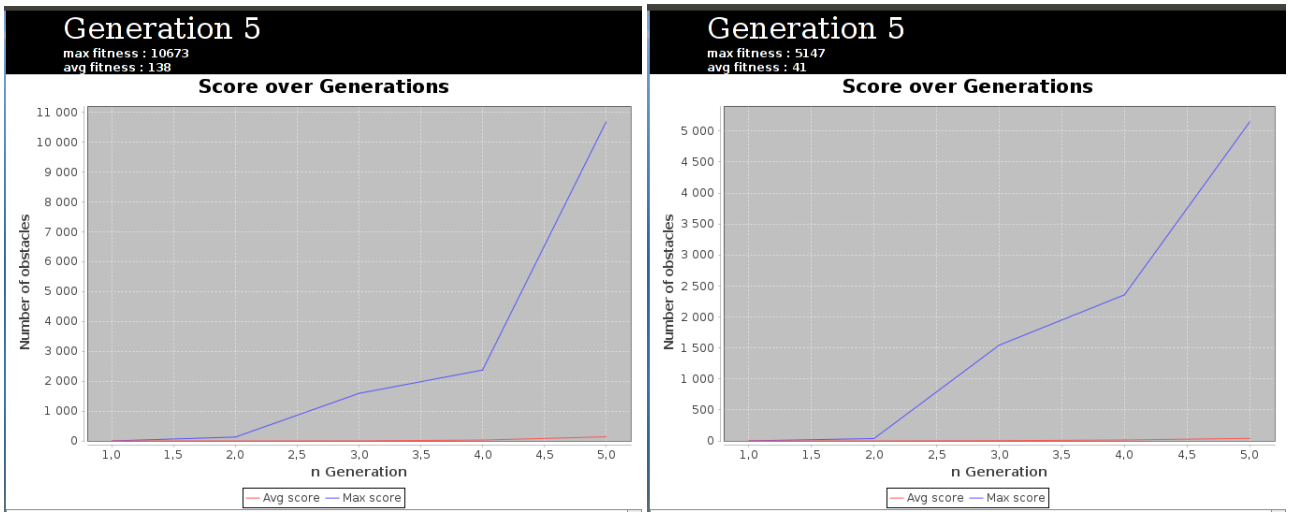


FIGURE 4 – Modèle ADN *NeuralNet* produit des résultats très satisfaisants en seulement 5 générations, pour $N = 1000$

Il est intéressant, bien que très complexe, de chercher à comprendre l'aptitude des réseaux de neurones à la résolution de ce problème.
En effet il est presque impossible de comprendre les valeurs d'un réseau de neurones, ce qui en fait une boîte noire peu engageante. Nous savons cependant qu'un réseau de neurones va plutôt chercher les relations et corrélations entre les input, plutôt que d'agir comme une mémoire brute. Dans notre cas, le comportement à adopter semble être une relation très simple de la position relative. Cela signifie que le réseau pourra très simplement la trouver - c'est donc l'approche du réseau qui justifie son très volumineux espace solution.

Il est tout de même essentiel de souligner qu'il est aussi fréquent que l'algorithme se bloque dans un minimum local (stagnation à 10000 obstacles) : la taille de l'espace est impressionnante mais il est jonché de minima locaux qui rendent l'exploration très aléatoire ; même si un résultat très satisfaisant est toujours rapidement obtenu.

3.4 Méthodes de sélection

Soit P une population, c'est-à-dire un tableau de N Individus. Un Individu est défini comme un couple $(fitness, ADN) \in \mathbb{R} \times \Theta = \mathcal{I}$. On admet que les Individus disposent de la capacité de *crossover*, c'est à dire de se coupler à un autre Individu pour en créer un nouveau - via une fonction que l'on peut définir comme $c : \mathcal{I}^2 \longrightarrow \mathcal{I}$. Cette opération mélange leur ADN et crée un enfant de $fitness = 0$.

Soit $\mathscr{P}$ l'ensemble des populations, on définit une *fonction de sélection* comme un fonction $sel : \mathscr{P} \longrightarrow \mathscr{P}$. Evidemment, beaucoup d'entres elles sont inintéressantes. L'objectif principal d'une fonction de sélection est qu'étant donnée une population P, elle ressorte une nouvelle population P' plus efficace que P. Pour ceci, seules les *fitness* des individus vont être d'utilité.

Pour générer un individu de la nouvelle population, nous choisissons d'utiliser la fonction de *crossover* de deux individus de P. Une autre manière de voir la fonction sel est alors de dire qu'elle associe, à un tableau de fitness, un tableau de probabilités de même taille. La probabilité de chaque indice du tableau correspondra alors à la probabilité que l'individu du même indice dans P soit sélectionné pour être parent d'un individu de la nouvelle population. Nous pouvons alors définir $sel' : \mathbb{R}^N \longrightarrow [0, 1]^N$ une fonction qui sera appelée par la fonction générique sel pour faire l'essentiel du travail à sa place. Le travail de sel sera simplement d'appeler sel' puis de créer N nouveaux individus en sélectionnant, grâce au tableau de probabilités, deux individus de P pour *crossover*.

Etudions maintenant différentes fonctions sel' possibles. Nous noterons $f = (f_1...f_N)$ le vecteur fitness d'entrée de sel' et $p = (p_1...p_N)$ son vecteur de probabilité de sortie.
Nous testerons ces fonctions avec le modèle ADN *BoolArray*, comme il est plus prévisible que *NeuralNet*.

3.4.1 BasicSelection

La première intuition est d'associer une probabilité plus forte aux individus de meilleure fitness. La méthode la plus simple pour ceci est de simplement associer

$$p_i = \frac{f_i}{\sum f_i}, \ \forall i$$

C'est l'idée développée dans la première méthode de sélection que nous avons implémenté naïvement, *BasicSelection*.
Cette méthode est hautement inefficace. Elle implique qu'un individu ayant fait k fois mieux qu'un autre aura k fois plus de chances d'être sélectionné. Cependant, la population est de taille N, et on peut se douter que l'élite ne sera qu'une petite partie de la population. Autrement dit, il faut que k contrebalance N pour que l'élite ne se fonde pas dans la masse, ce qui représente une performance monstrueuse ! Généralement, le plus probable sera de piocher dans la majorité inefficace. L'algorithme ne converge alors pas, peu importe les hyperparamètres - au mieux,la population stagne.

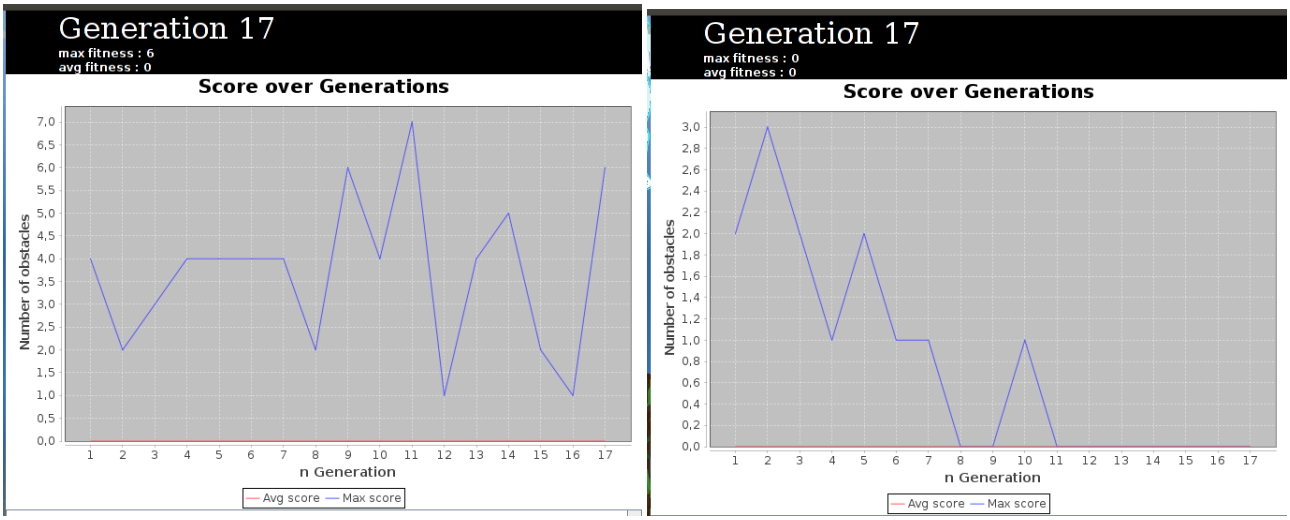


FIGURE 5 – Méthode de Sélection *Basic Selection* stagne à 3 ou 4 obstacles (élite) à gauche, empire à droite, pour $N = 1000$

3.4.2 FunctionalSelection

La raison principale de l'échec de la méthode précédente était le manque de poids de l'élite dans la population. Pour contrer ceci, nous pouvons appliquer une fonction qui augmentera considérablement les individus aux meilleurs scores face au reste. Intuitivement, il faut donc appliquer une fonction croissante convexe aux fitness pour creuser l'écart entre les individus. Informatiquement, il n'est pas pratique d'appliquer cette fonction directement sur les *fitness*, car leur valeur peut être grande - si on considère une fonction exponentielle, les valeurs risquent de rapidement exploser les plafonds des types Java.
Il est plus judicieux de d'abord restreindre les valeurs sur $[0, 1]$. Pour ceci, nous pouvons simplement réhausser les valeurs au dessus de 0 (par simple soustraction par $minf_i$ si elle est négative), puis normaliser en divisant par $maxf_i$. Nous pouvons alors appliquer une fonction positive, convexe et croissante f_c qui passe par $(1, 1)$ et les valeurs seront contenues dans $[0, 1]$. Qualitativement, les valeurs des fitness des individus les moins performants sont rabaissées pour donner du poids aux plus hautes valeurs.
Les fonctions convexes implémentées sont x^p (sachant que $p = 1$ donne le même résultat que *BasicSelection* et $e^{\alpha(x-1)}$

Cette méthode, bien que meilleure que la première, reste peu performante. En effet, si les individus ont des scores assez proches, l'effet ne sera pas ressenti à moins que la pente au voisinage de 1 soit très forte.

Dans ce cas, si un individu mieux que le lot, il écrasera le reste et sera pratiquement le seul élu à la sélection pour *crossover*. Ceci aura pour effet de briser la diversité du génome, ce qui ne permettra de converger que si l'individu était déjà dans l'espace solution. La convergence occurera en cas de fragile équilibre entre le choix des hyperparamètres de f_c et le processus stochastique.
Les résultats ci-dessous sont pour deux exécutions consécutives de $f_c(x) = x^5$, qui offre les résultats les plus convaincants :

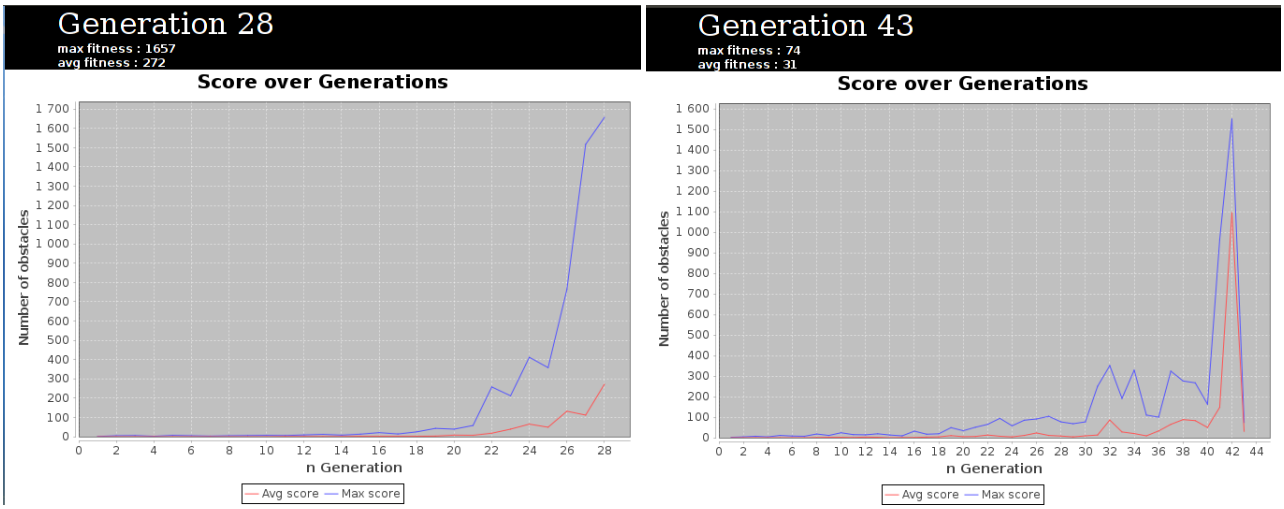


FIGURE 6 – Selection *FunctionalSelection* converge d'abord, puis dégénère à l'exécution suivante.

On observe clairement les effets stochastiques sur les résultats ; cette méthode n'est donc pas stable.

3.4.3 RankSelection

Pour nous séparer des problèmes de valeurs relatives des *fitness*, nous avons implémenté une méthode qui n'en tient pas du tout compte. L'idée est très simple : sélectionner les individus en fonction de leur rang dans la population ; c'est-à-dire que l'on trie les Individus de manière décroissante selon leur fitness. On décide ensuite d'un hyperparamètre : la probabilité que le premier (meilleur) individu soit sélectionné, notée r_p (*rankSelectionProba* dans le code).
L'algorithme de sélection est alors très simple : on teste d'abord si le premier individu est sélectionné pour être parent (donc avec probabilité r_p). Sinon, on teste la même chose pour le second, et ainsi de suite. Cela signifie que l'individu i dans le tableau trié a pour probabilité $p_i = (1 - r_p)^{i-1} r_p$ d'être sélectionné. Le dernier individu est sélectionné par défaut si aucun autre ne l'a été avant lui.

Comme explicité dans les motivations, cette méthode permet de palier à plusieurs des problèmes soulevés précédemment. Nous pouvons tout de même en soulever des défauts. En effet, on observe que la probabilité d'être choisi décroît exponentiellement avec i. Cela signifie que pour un N un minimum large (et r_p raisonnable), les individus de la fin du tableau ont une probabilité quasi inexistante d'être sélectionnés. Cela peut porter atteinte à la diversité du génome, cependant un choix "sage" de r_p permet de contrôler qu'un certain pourcentage de la population sera appelé suffisemment de fois, et donc d'établir une gestion de cette perte de diversité.

Les résultats sont, tout simplement, tous les résultats présentés avec les modèles ADN du début de la section. En d'autres termes, cette méthode est responsable de tous nos résultats les plus encourageants.

3.5 Etude sur les hyperparamètres

Nous avons déjà décrit une grande partie des hyperparamètres : les différents modèles ADN, les différentes méthodes de sélection, et leur propres hyperparamètres. Nous n'essairons pas de modifier la fonction de *fitness*, car le problème est assez simple. Les hyperparamètres intéressants que nous n'avons pas étudié en profondeur sont alors

— La taille de la population N.

— L'amplitude des mutations $mutAmpl$, et sa capacité à décroître.
— La probabilité r_p de sélection de rang de *RankSelection*, dont nous n'avons pas fourni d'analyse.

Pour l'observation des résultats, nous choisirons encore *BoolArray* pour sa prévisibilité, et *RankSelection* pour son efficacité.

3.5.1 Taille de la population

Le nombre d'individus dans la population est fixé avant l'exécution de l'algorithme, et va impacter beaucoup d'autres hyperparamètres ensuite.

Intuitivement, nous devinons qu'une grande population permettra d'avoir un génome plus diversifié, mais aussi un temps d'exécution plus lent. A l'inverse, une petite population sera beaucoup moins stable face aux effets stochastiques de l'algorithme : un individu très adapté à une unique situation pourra avoir beaucoup plus de poids. Pire, une génération de mauvaise qualité (à cause, par exemple, de mutations néfastes) ne pourra pas se rétablir : en effet si les individus effectuent tous un mauvais score, il sera impossible de départager les individus avec le meilleur ADN.

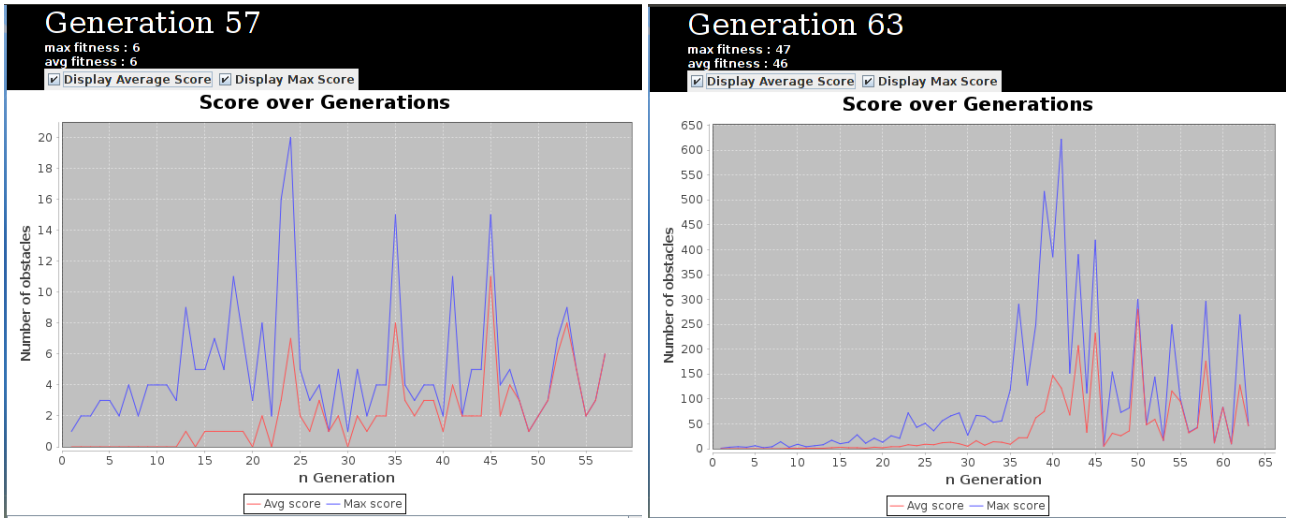


FIGURE 7 – A gauche, $N = 50$; à droite, $N = 250$

Les valeurs de $rankSelectProba$ ont été adaptées (réhaussées) pour ces exécutions.
On observe que pour $N = 50$ la population n'est pas suffisemment diversifiée pour trouver un comportement intéressant, et les mutations ne permettent pas d'explorer suffisemment l'espace. Pour $N = 250$ l'algorithme était proche de la convergence mais on observe de forts effets stochastiques qui ont finalement eu raison de l'algorithme qui n'a pas réussi à trouver de comportement suffisemment intéressant avant que les mutations ne diminuent trop.

3.5.2 Amplitude des mutations

Comme énoncé précédemment, les mutations permettent d'explorer le voisinage d'un point $ADN \in \Theta$. Le rayon de ce voisinage va être donné, dans le code, par la valeur de $mutAmpl$, l'amplitude des mutations [14].
Cette valeur a beaucoup de sens dans un contexte continu, mais n'est pas applicable au domaine discret des *BoolArray*.

Il est très difficile de deviner une valeur acceptable pour l'amplitude de mutation. Le problème principal est que celle-ci est sensée pouvoir répondre à trois problèmes contradictoires :

1. Extraire un individu d'un minimum local - et donc être suffisemment grande pour sortir de son voisinage.

14. Notons que ce voisinage trace plutôt un hypercube de côté $mutAmpl$.

2. Explorer Θ dans les zones inconnues (où les individus ne sont jamais allés, et donc le *crossover*, ne peut aller car copie des valeurs existantes).
3. Explorer le voisinage du point ADN lorsqu'il se rapproche de ADN^*.

En clair, $mutAmpl$ doit être à la fois suffisemment grande pour permettre une exploration profonde de Θ, mais suffisemment faible pour s'approcher du point solution sans le dépasser.

Pour palier à ces deux conditions incompatibles, il est possible de passer par une heuristique. L'idée est que l'exploration n'est vraiment nécessaire qu'au début de l'algorithme, lorsque les individus sont générés aléatoirement et sont donc répartis de manière équilibrée dans Θ. Par contre, l'exploration du voisinage n'est vraiment nécessaire que lorsque l'algorithme a déjà engendré quelques générations et que celles-ci se rapprochent de l'espace solution.

En d'autres termes, il est possible de répondre à ces deux problèmes en faisant décroître l'amplitude de mutation avec le temps, c'est-à-dire les générations. Notre but étant que ces dernières tendent vers 0 (ou une valeur choisie $minValue$) avec les générations. La valeur pour la génération 0 est donnée par $mutationAtZero$. Nous devions donc choisir une fonction convexe décroissante passant par $(0, mutationAtZero)$ et tendant vers $minValue$. Ces conditions sont lâches et on peut en plus imposer une vitesse de décroissance ; autrement dit, fixer un autre point $(val, mutationAtVal)$. Nous avons choisi une fonction exponentielle décroissante de la forme $\alpha e^{-\beta x} + minValue$.
Il est important de souligner qu'il faut avoir une idée du temps de convergence de la méthode pour pouvoir donner la valeur $(val, mutationAtVal)$.

Cette méthode a tout de même été implémentée pour les *BoolArray*, en remplaçant simplement $mutProba$ (alors inutile) par $mutAmpl$. Cela permet souvent de faire la différence capitale entre la convergence et la stagnation, dûe à des aller-retours trop fréquents empêchant d'avancer.

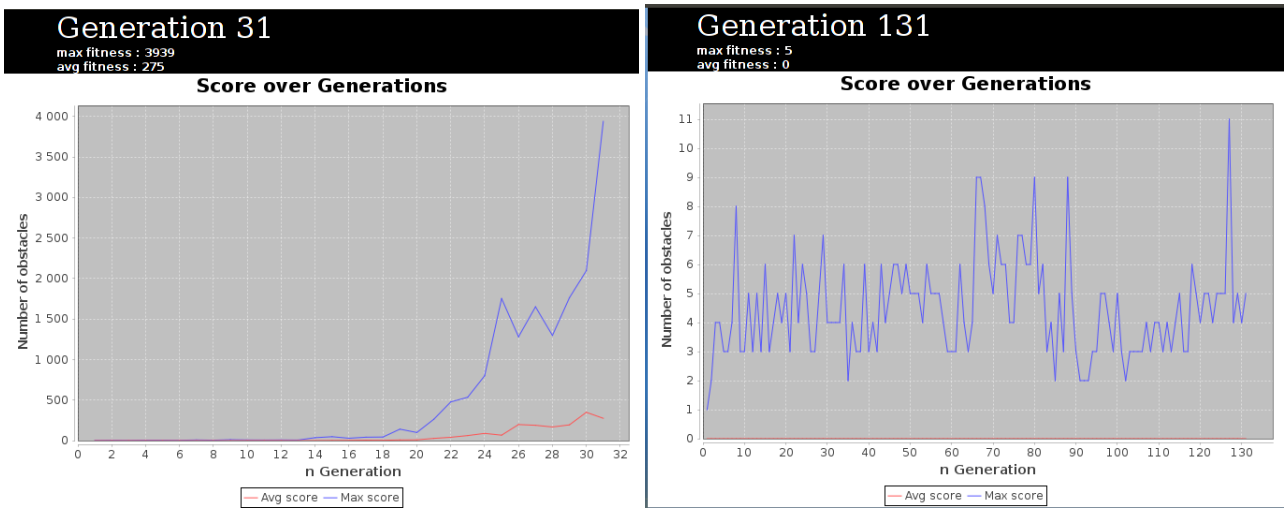


FIGURE 8 – Décroissance des mutations avec le temps : TRUE à gauche, FALSE à droite

3.5.3 rankSelectProba

Nous avons vu que la méthode de sélection *RankSelection* était de loin la plus performante. Ceci dit, elle ne l'est que sur un équilibre délicat de son paramètre $r_p = rankSelectProba$.
Celui-ci donne l'importance qu'il faut donner à l'élite. S'il est trop imporant, on brise rapidement la diversité, et des individus qui ne sont que l'élite immédiate finissent par être beaucoup trop représentés et donc leur erreurs commises à répétition.
Dans l'autre sens, si le paramètre est trop faible, la convergence est très lente, ce qui n'est pas compatible avec la méthode de décroissance de mutations vue précédemment.

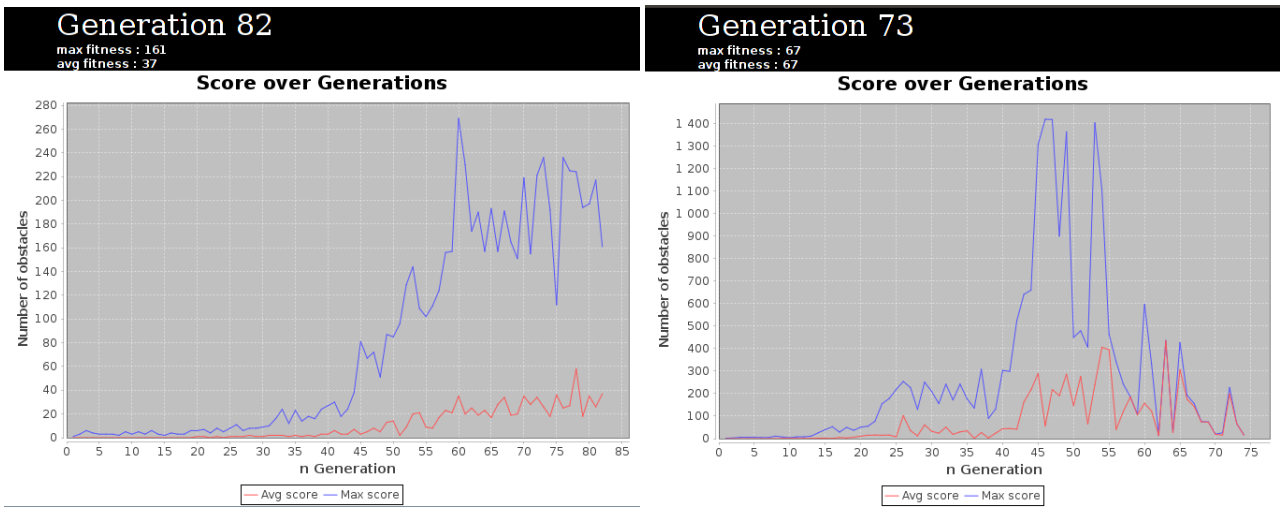


FIGURE 9 – A gauche, $rankSelectProba = 0.005$ (trop faible) ; à droite, $rankSelectProba = 0.15$ (trop fort)

On observe que ces deux valeurs 0.005 et 0.15 ont toutes deux offert des individus intéressants puisqu'ils ont dépassé respectivement les 250 et 1300 obstacles ; cependant, ils n'ont pas réussi à être suffisemment stables pour converger avant que les mutations deviennent négligeables (environ 50 générations).
Cela montre qu'il existe une fenêtre assez maigre dans laquelle prendre $rankSelectProba$ pour que *RankSelection* soit très performante.

4 Déroulement du projet

La première étape de ce projet a été d'implementer notre version revisitée du jeu "Flappy Bird" en Java. Nous nous sommes appliqués à respecter la structure Modèle-Vue-Contrôleur. Cela s'est avéré très utile pour la suite du projet qui a rapidement pris de l'ampleur. Mais surtout l'ajout de la partie Intelligence Artificielle a été plus intuitive grace à cela. Voici le diagramme de classes associé au simple jeu :

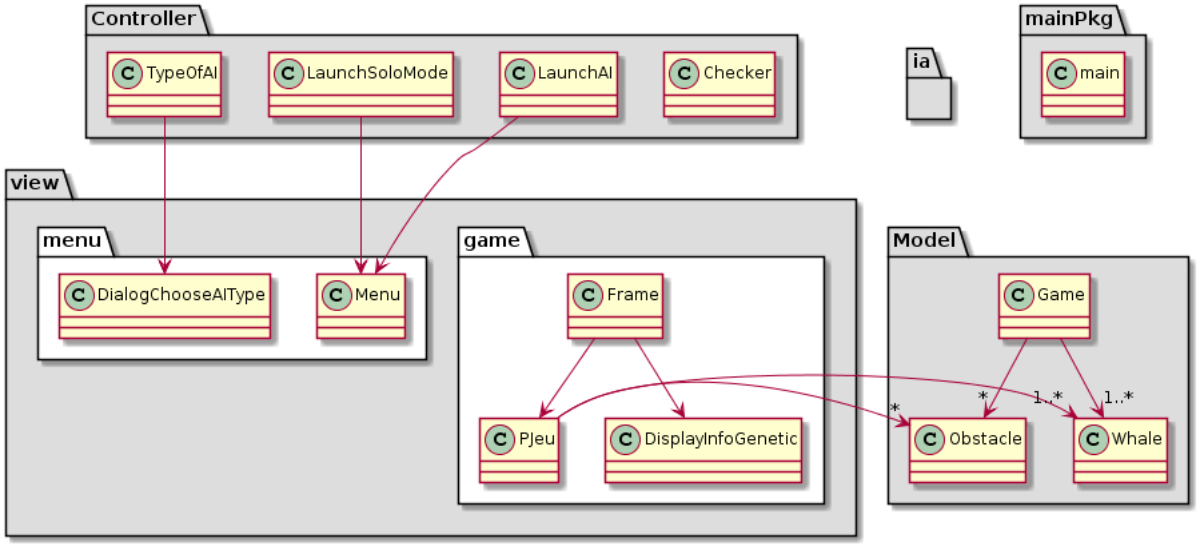


FIGURE 10 – Diagramme de classes du jeu

Après une période de documentation sur les différents concepts, nous avons réfléchi à la manière de les appliquer à notre problème. C'est ainsi que nous avons identifié 2 combinaisons possibles (i.e. algorithme génétique/maillage et algorithme génétique/réseau de neurones).

Dans un premier temps, nous avons implémenté l'algorithme génétique avec le maillage car cette solution était plus simple (bien que plus naïve).

Dans un second temps et avec plus de difficultés, nous avons implementé le réseau de neurones. Et pour terminer nous avons réalisé des ajouts/modifications afin de rendre les intelligences plus efficaces et plus rapides à l'apprentissage. Pour cela nous avons recherché et testé d'autres fonctions et hyperparamètres. Nous avons, avant cela, restructuré le projet deux fois afin de faciliter le changement des paramètres et rendre le projet plus pérenne. Voici la structure du package ia à la première et seconde restructuration :

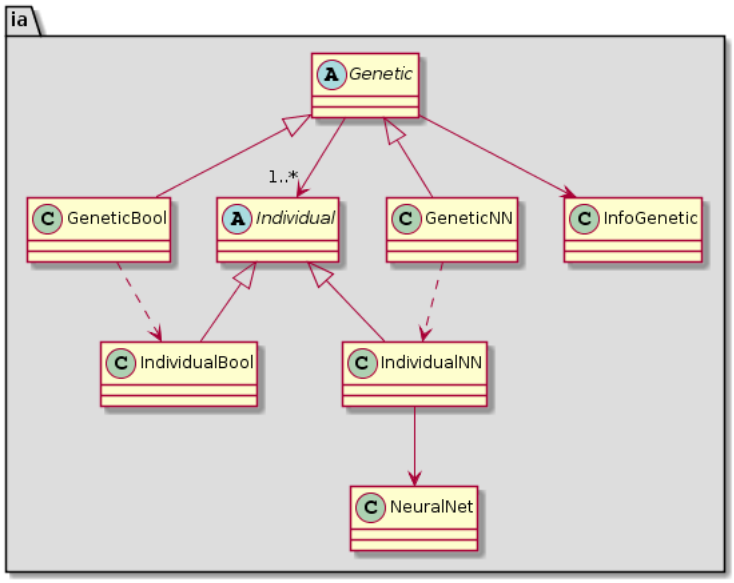


FIGURE 11 – Première restructuration

Ce modèle cherchait à traduire le fait qu'un Individu et l'algorithme Génétique avaient des éléments en commun peu importe les modèles choisis, d'où l'idée d'en faire des classes abstraites. Cependant, nous voulions faciliter au maximum la tâche à une personne souhaitant tester un nouveau modèle ADN. Dans ce cas, l'ajout d'un modèle ADN implique aussi de créer une nouvelle classe qui étend Genetic et une nouvelle classe qui étend Individual, en plus du codage de son modèle ADN. C'est ainsi que nous sommes arrivé à cette seconde reconstruction :

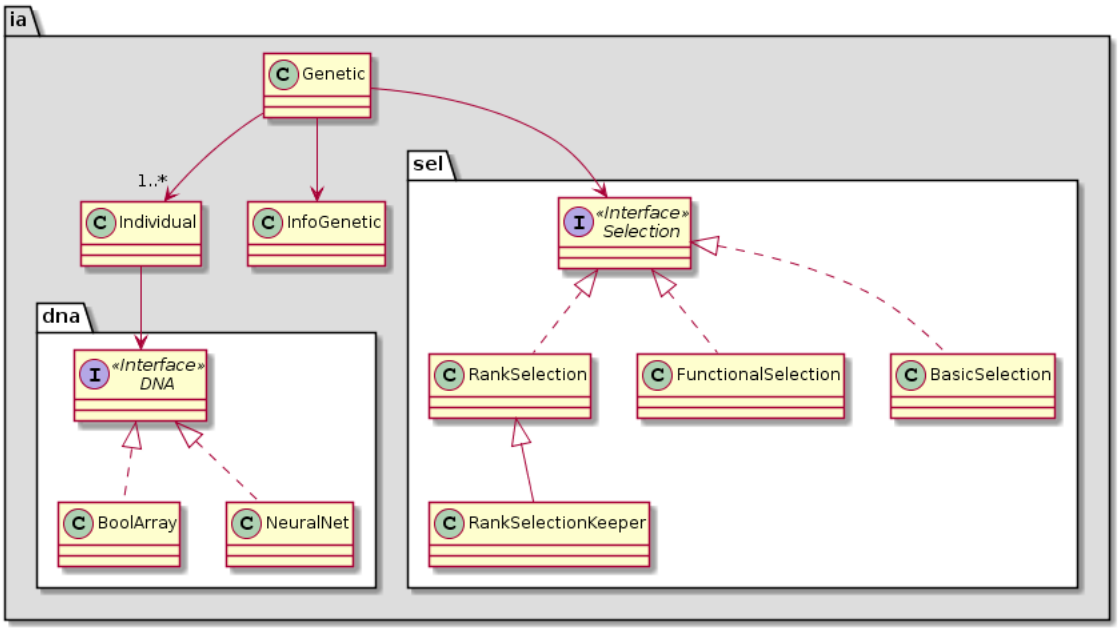


FIGURE 12 – Seconde restructuration

Dans ce modèle, l'ADN est géré comme une Interface - une modèle ADN doit avoir sa *fonction de décision* $f_{E,ADN}$, ainsi qu'être capable de *crossover* (fonction notée c plus haut) avec un autre ADN pour fournir l'ADN d'un nouvel enfant. Ces deux fonctions constituent l'Interface *DNA*. L'ajout d'un nouveau modèle ADN demande donc seulement d'implémenter ces deux fonctions.

Afin de continuer dans cette lancée nous en avons fait de même pour la fonction de sélection ; il suffit alors de créer une classe implémentant la fonction de sélection de l'Interface *Selection* pour pouvoir l'utiliser dans l'algorithme.

5 Conclusion

Ce projet nous a permis d'approfondir la programmation par objet. En effet, nous avons appris a bien structurer un projet afin de le rendre plus pérenne et plus intelligible. Cela se reflète par le respect de la structure Modèle-Vue-Controleur et par l'utilisation d'interface pour permettre le changement de paramètres et de type d'IA.

Nous avons découvert les concepts d'algorithme génétique et de réseau de neurones, notions qui ont ensuite été implementées et testées sur notre problème.

Nous avons été agréablement surpris de voir que les méthodes permettent d'obtenir des individus très performant en très peu de temps d'exécution.

Néanmoins, nous nous sommes rendu compte que d'une exécution à l'autre les variations dans les résultats peuvent avoir une amplitude très importante. C'est pourquoi nous avons compris que le choix du modèle de départ (maillage ou réseau de neurone) est important, tout comme le choix des hyperparamètres.

Concernant les modèles, le maillage offre des résultats corrects dans un temps raisonnablement court. Quant au réseau de neurones, il offre des résultats bien plus élévés en terme de *fitness* mais a l'inconvenient d'être plus lent car la réponse du réseau nécessite un certain temps de calcul.

Pour le maillage, une amélioration pertinante serait d'utiliser des valeurs continues sur $[0, 1]$ et décider de sauter si la valeur est $> 0,5$. Il serait alors possible d'utiliser le principe de mutations d'une génération à l'autre en fixant une proba de mutation comme pour le réseau de neurones. Ceci aurait permis de travailler dans un espace continu, bien plus adapté à l'algorithme.

Notre projet permet de fournir un individu fort pour l'environnement en question. Mais une fois selectionné, il ne pourra pas s'adapter à un nouvel environnement. L'étape suivante serait naturellement d'essayer d'autres techniques d'apprentissage par renforcement, plus abouties, qui permettraient d'avoir un seul individu qui apprendrait au fur à mesure qu'il effectue des erreurs.

Nous sommes tout particulièrement contents et reconnaissants d'avoir pu travailler sur un sujet de notre choix, ce qui nous a permis de nous y plonger plus encore et de se mettre au défi.

6 Annexes

Veuillez trouvez ci-dessous les différents diagrammes de classe correspondant à notre implémentation.

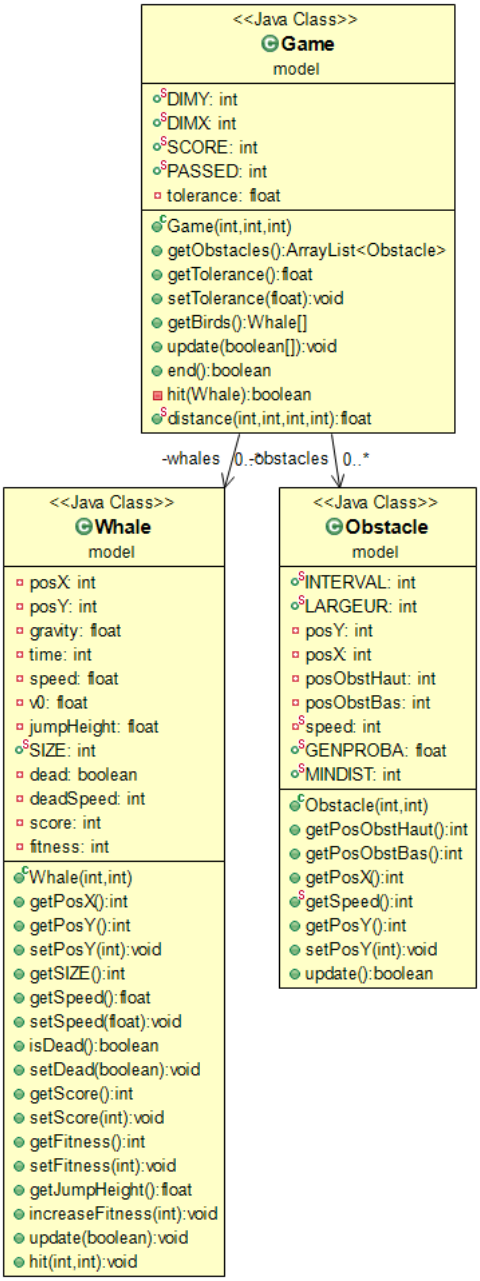


FIGURE 13 – Modèle

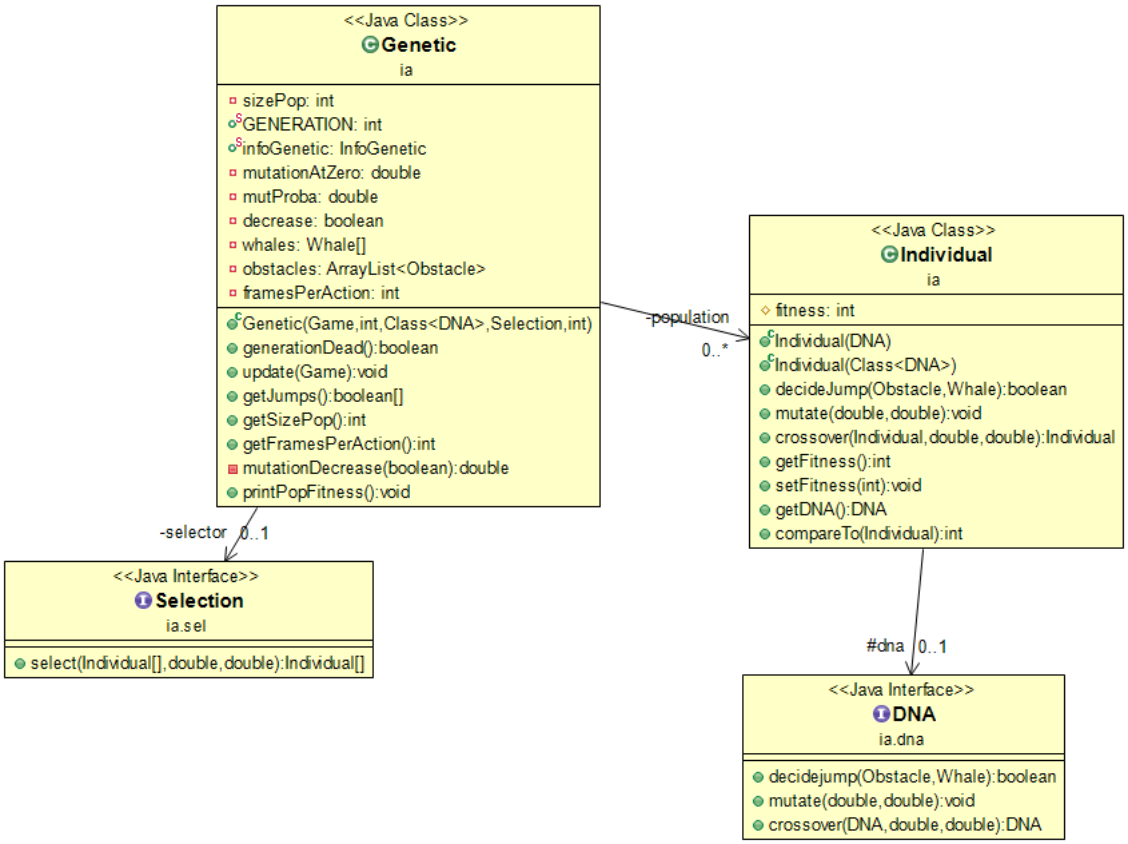


FIGURE 14 – Architecture du package IA

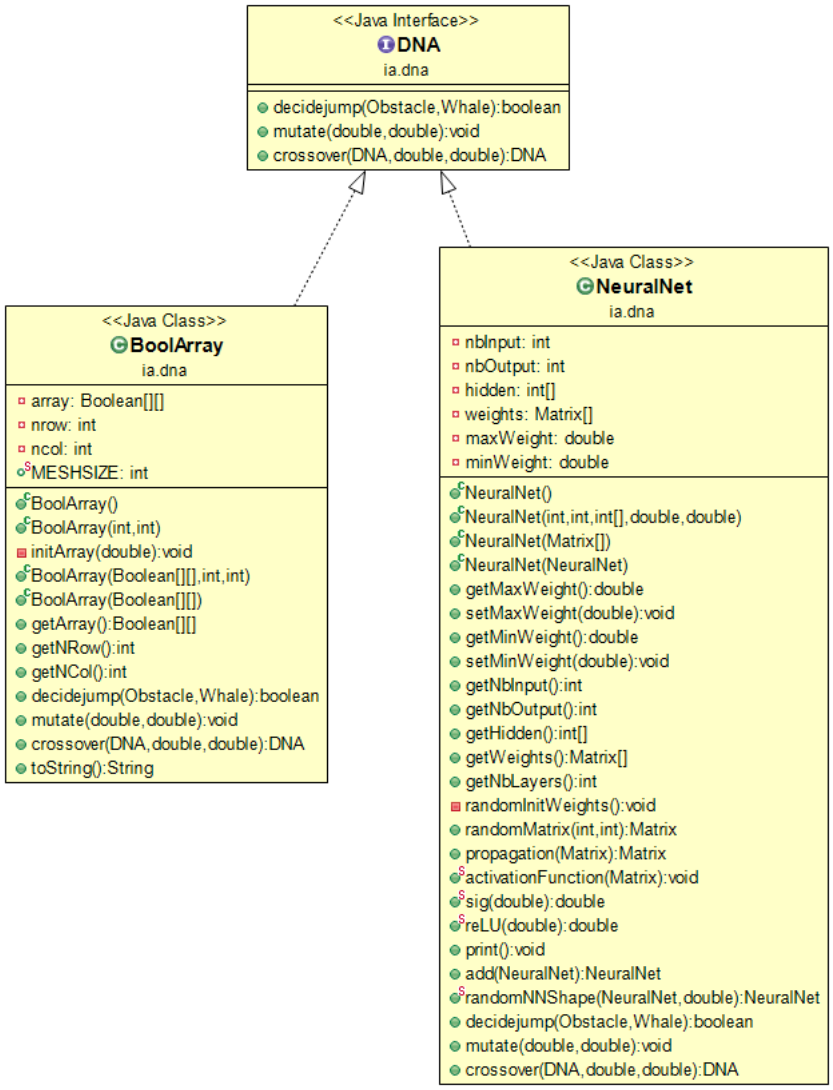


FIGURE 15 – package DNA

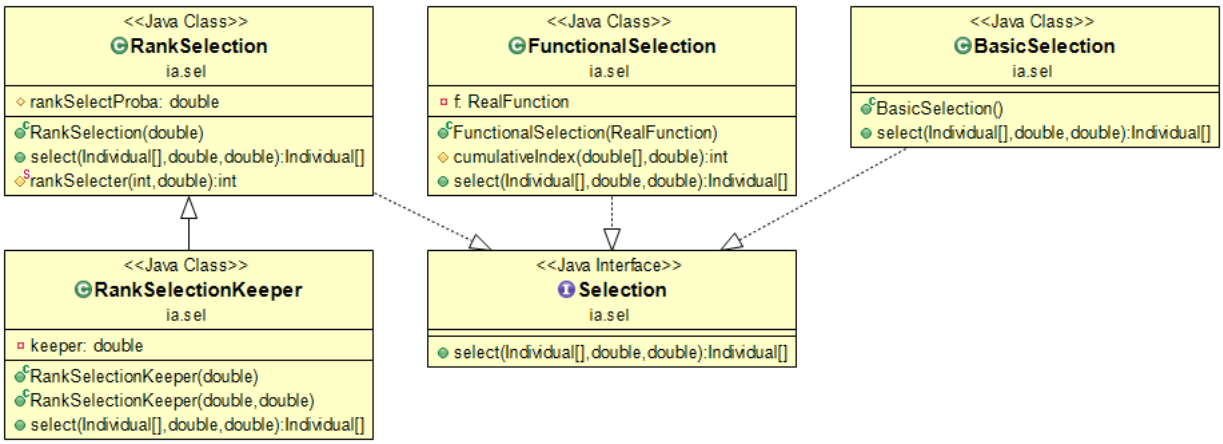


FIGURE 16 – package Selection