Rapport Projet - ANDROIDE

Conception de comportements pour ROBOBO

Encadrant : Stéphane Doncieux

Martine Girard
Maximilien Groult
Quentin Mathias
Tom Portoleau

Table des matières

1 Cahier des charges

Introduction

Un algorithme évolutionniste est un algorithme de recherche opérationnelle basé sur la théorie de l'évolution. Le principe est le suivant : une population de base est créée, puis, on génère une nouvelle population, dont chaque individu hérite d'une partie des caractéristiques de ses parents. La nouvelle population est sélectionnée en évaluant les nouveaux individus par une fonction objectif. On réitère le procédé jusqu'à atteindre une solution (individu) satisfaisante. Le problème de cette idée est qu'elle ne permet pas de s'extraire des extrema locaux[(2)].

C'est pourquoi Lehman et Stanley ont proposé de rechercher des solutions sans se préoccuper de l'objectif. Ce procédé s'appelle NS (Novelty Search). A chaque génération, on concentre la recherche sur les individus présentant le plus de nouveauté. Cette nouveauté est calculée par une fonction d'évaluation, basée sur une fonction de distance entre deux individus.

Néanmoins, les algorithmes basés sur NS présentent l'inconvénient de se perdre lorsque l'espace des solutions est trop vaste. La méthode IEC (Interactive Evolutionary Computation) permet de pallier cela. Cette nouvelle idée utilise l'intuition humaine afin d'orienter les algorithmes évolutionniste. Toutefois, un humain possède une endurance limité et fini par se fatiguer, ce qui empêche donc cette méthode d'être efficace sur un grand nombre d'itération.

L'approche NA-IEC (Novelty-Assisted Interactive Evolutionary Computation) propose de fusionner les deux méthodes précédentes et de régler ce dernier problème. En effet, le Novelty Search permet de créer des individus suffisamment originaux en quelques itérations, et IEC les propose à un utilisateur. C'est cette idée que nous allons utiliser pour créer des comportements pour robot[(1)].

Le robot que nous allons utiliser, est ROBOBO. Ce robot a été créé par la société MyTechia. Ces robots sont nativement équipés de deux roues, de capteurs de distance et d'un socle permettant d'y placer un smartphone. Ces robots peuvent être connectés via Bluetooth à un smartphone qui peut lui offrir des capteurs supplémentaires (caméra, microphone, accéléromètre, etc...) et de la puissance de calcul. Le robot est ainsi piloté par une application Android sur le smartphone qui communique avec le robot par BlueTooth.

1.1 Choix d'implémentation

1.1.1 Jeu de paramètres vs Réseau de neurones

Le réseau de neurones est la méthode utilisée par Woolley et Stanley(1) et présente plusieurs avantages, dont la possibilité de générer des comportements imprévus. Une implémentation plus simple serait d'utiliser un jeu de paramètres fini, qui servirait à caractériser des propriétés d'un comportement (comme les différents mouvements utilisables, la vitesse d'exécution, la durée, l'amplitude des mouvements, etc)

Cette approche est plus restrictive mais nous permet de mieux observer l'évolution des comportements et de comprendre plus facilement les problèmes qui pourraient potentiellement être rencontrés.

Pour ces raisons nous avons décidé de choisir l'implémentation des comportements par un jeu de paramètres car c'est un moyen simple et efficace pour représenter des comportements simples.

1.1.2 L'algorithme génétique

Mutation : implémentée dès le début, permet de générer de nouveaux individus qui seront sélectionnés par la suite . On commencera par une sélection en fonction des différences de paramètres, avant de s'orienter vers NS [(2)].

La sélection : NS + humain. Pour implémenter NS il nous faut déjà avoir un système de mutation fonctionnel c'est pourquoi nous nous en occupons en second. L'idée est aussi que nous pourrons aisément vérifier le bon fonctionnement de la présélection NS grâce au fait que nous travaillerons (à ce moment là) avec un jeu de paramètre relativement simple (par exemple par représentation graphique des populations générées puis filtrée par NS). Pour cela, nous aurons à définir une méthode de comparaison de comportements qui nous permettra de mesurer l'originalité des nouveaux individus.

Crossover : implémenté par la suite, après s'être assuré du bon fonctionnement de NS et de la mutation sur des individus (et non des couples).

1.1.3 L'utilisation du robot

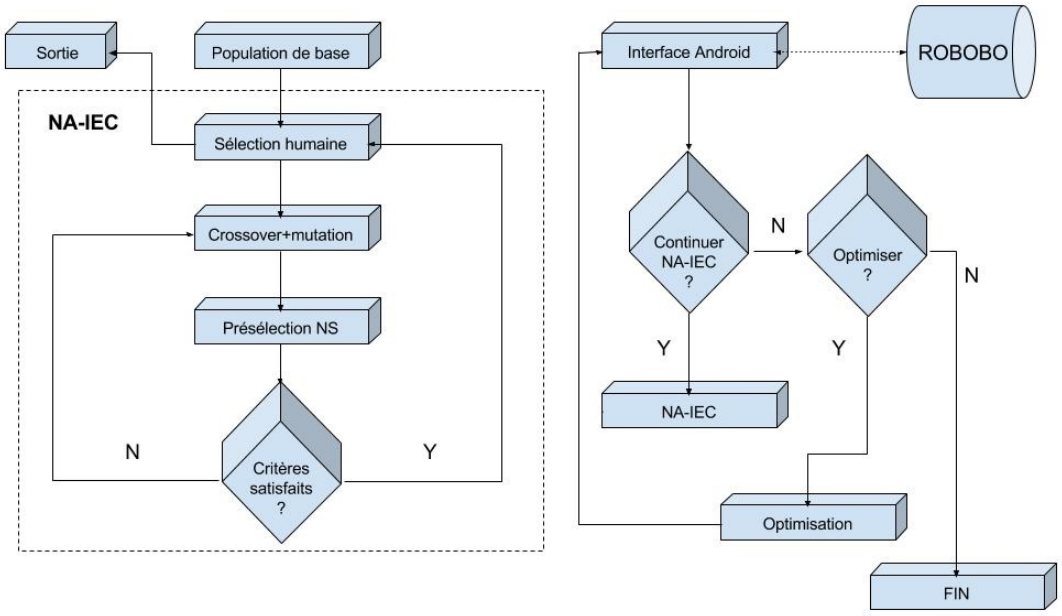
Pour commencer nous utiliserons pas les capteurs du robots. Par la suite on pourrait l'envisager, pour faire du suivi de lumière ou éviter des obstacles par exemple. On pourra également se servir du microphone et de la caméra du smartphone pour implémenter des comportements de bases originaux.

1.1.4 Codes sources éventuellement utiles

- Le Dynamic Time warping va permettre de mesurer la similarité entre deux comportements.
- Framework C++
- Sferes2 [(3)]

1.1.5 Organigramme

Voici l' organigramme qui résume ce que nous allons implémenter :



1.2 Estimation de coût

Nom de la fonctionnalité	Temps estimés	Personnes
Interface de base android	1-2 semaines	Martine Maximilien Quentin
Interface avancée android	2-4 semaines	Martine Maximilien
Mutations	1 semaine	Maximilien Tom Quentin
Sélection humaine	1 semaine	Martine Quentin
NS	1-2 semaines	Tom Quentin Maximilien
Crossover	1-2 semaines	Maximilien Martine Tom
Jeu de paramètres et comportement de base	1-2 semaines	Martine Tom
Réseau de neurones -simulateur	2-3 semaines	Tom Maximilien

2 Manuel utilisateur

Voici une description simple de l'application que nous avons mis au point, et de comment s'en servir.

Écran principal

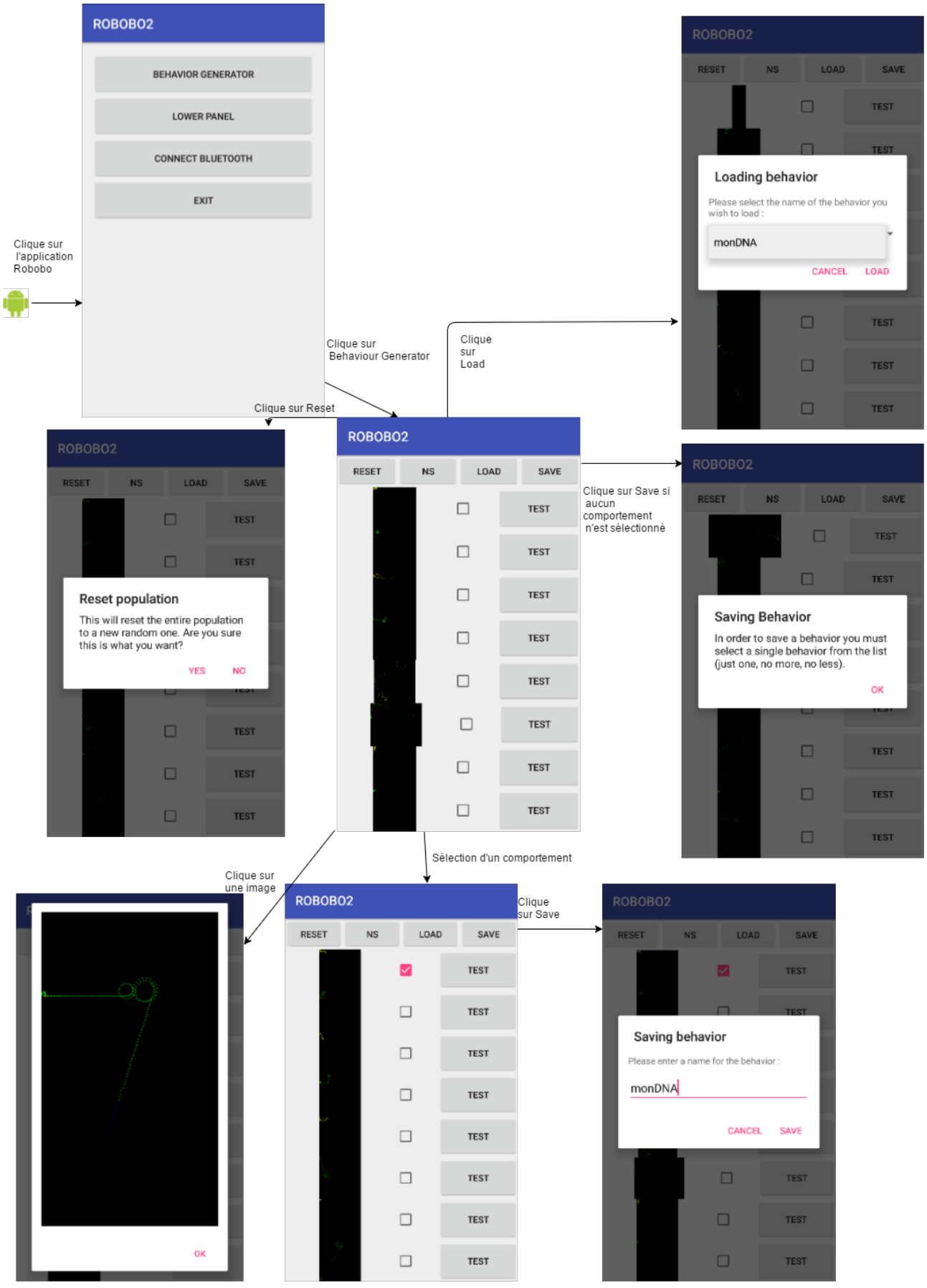
- Connect Bluetooth : Fait apparaître une nouvelle fenêtre, qui permet de choisir à quel ROBOBO se connecter. Si la connexion échoue, il faut relancer l'application.
- Run Test : Permet de baisser le panneau sur Robobo. A utiliser avant d'éteindre le robot.
- Behavior generator : Permet de générer les comportements. Amène vers l'écran de génération des comportements. Un Robobo doit être connecté avant d'appuyer sur ce bouton.

Écran de génération des comportements

- Reset : Génère une nouvelle population totalement aléatoire, et efface la population précédente.
- NS : Apres avoir selectionné au moins deux comportements dans la liste des comportements affichés , créer une nouvelle population, générée grâce à la Novelty Search.
- Load : Permet de charger un comportement sauvegardé au préalable, et l'ajouter à la population actuelle.
- End : Après avoir sélectionné un unique comportement, ouvre une fenêtre qui permet de le nommer, puis le sauvegarder. Il pourra être rouvert plus tard.
- Test : Permet de tester le comportement associé sur le Robobo. La trajectoire peut être observé sur l'image à gauche.
- Photo : Appuyer sur une photo permet de l'agrandir.

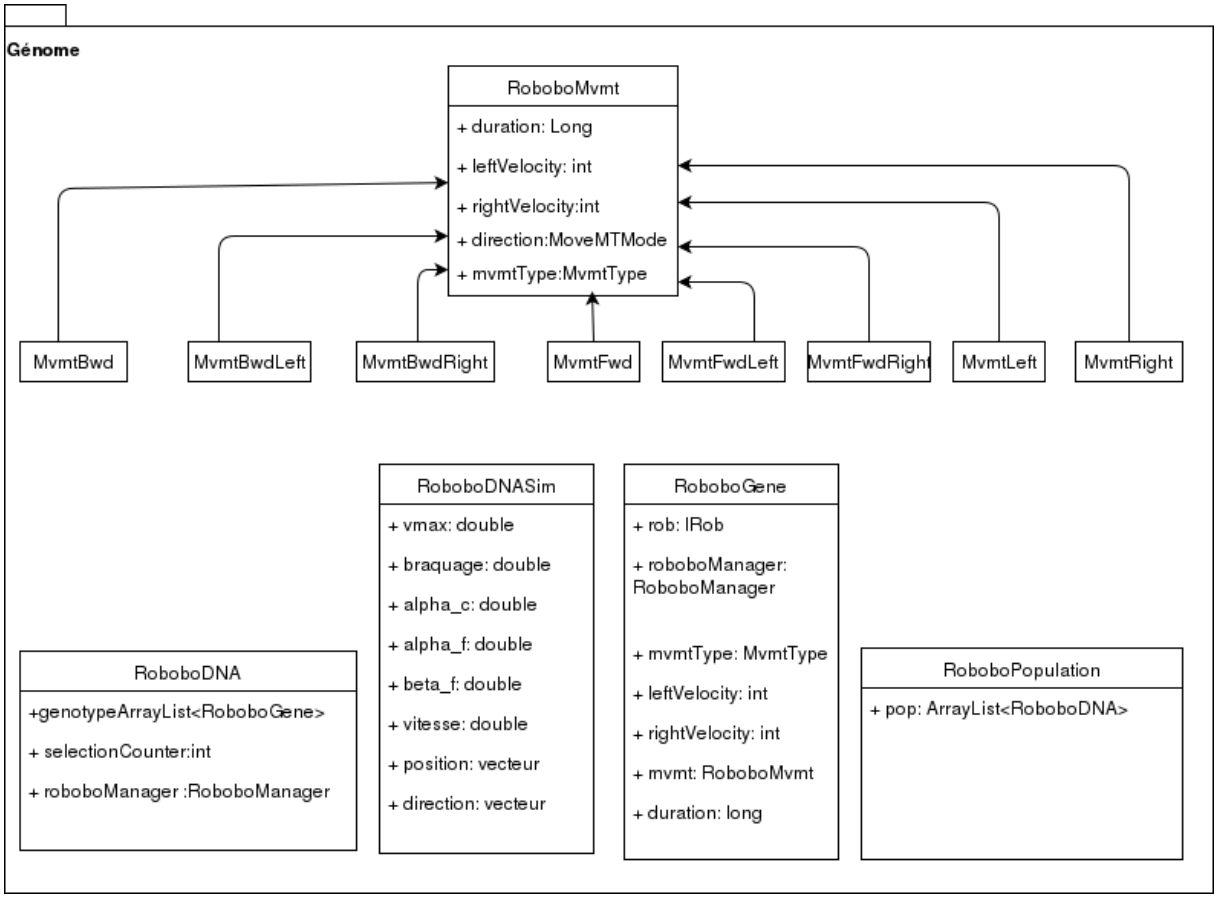
Appuyer sur la touche retour du téléphone permet de retourner au menu précédent sans perdre la population en cours.

3 Interface Homme-Machine



4 Architecture

4.1 Diagramme des classes



Package Genome

La package génome contient toutes les classes concernant la description et l'évolution des comportements de ROBOBO :

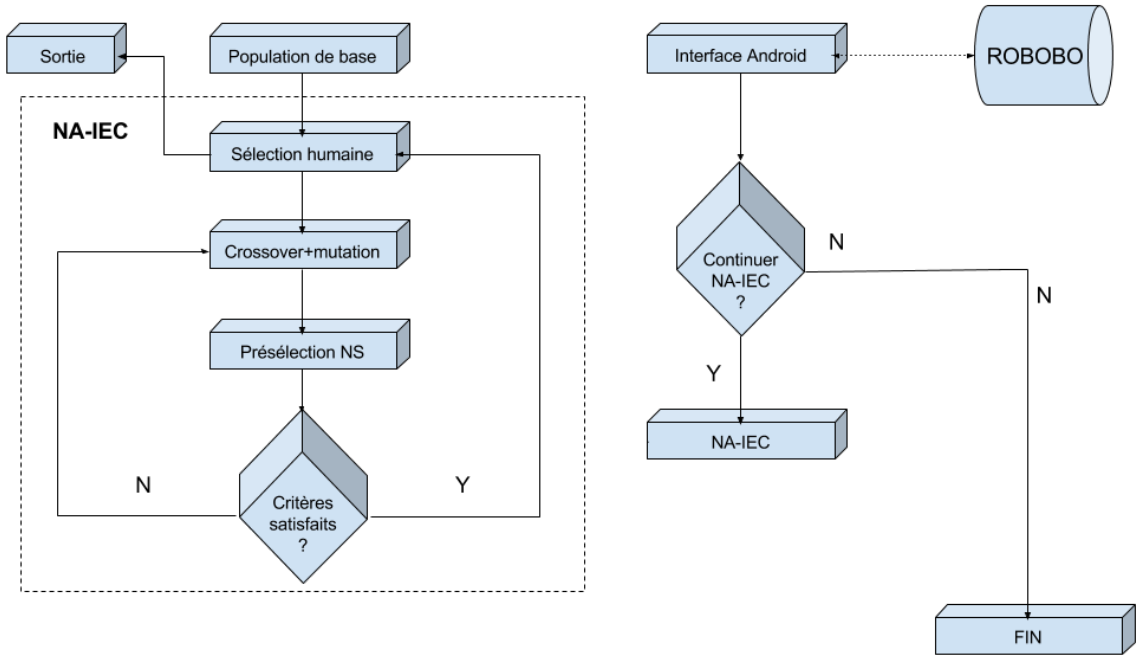
- **RoboboMvmt** : C'est dans cette classe abstraite que sont paramétrés les mouvements du robot. Par défaut leur durées sont de 200 millisecondes et , en cas de rotation, l'angle apporté est de 80 degrés. Les autres classes qui héritent de cette classe correspondent aux mouvements les plus simples (une unique instruction) que ROBOBO peut effectuer, tels que avancer (MvmtFwd), tourner à gauche (MvmtLeft) ou encore reculer en tournant à droite (MvmtBwdRight).
- **RoboDNAGene** : Cette classe représente un gène, qui correspondra à un unique mouvement.
- **RoboboDNA** : On modélise ici un "ADN" par une liste de gène. Ainsi, un ADN représente exactement un comportement de ROBOBO.
- **RoboDNASim** : Cette classe permet du simuler l'execution d'un ADN par ROBOBO. Elle nous sera utile pour prévisualiser les déplacements de ROBOBO dans l'espace sans avoir à effectivement le tester sur un robot.
- **RoboboPopulation** : Dans la mesure où un comportement est représenté par un ADN, une population sera représenté par une liste d'ADN. Ces populations seront impliquées dans le fonctionnement d'algorithme évolutionnaires.

Autres

- **MainActivity** : C'est la classe principale de l'application. Elle gère toutes les interactions utilisateur-application et application-ROBOBO.
- **RoboboApp** : C'est la classe qui nous a permis de faire nos experimentation, et qui sert maintenant à baisser le panneau de ROBOBO.

4.2 Organigramme

Voici l'organigramme décrivant le fonctionnement de notre application.



On peut remarquer que la partie optimisation qui apparait dans le cahier des charges n'a pas été implementée, car il ne nous a pas semblé pertinant de modéliser notre but par une fonction objectif.

5 Algorithmes Évolutionnistes

5.1 Algorithmes et choix

Phenotype / Genotype

Le phénotype est le comportement observé du robot, il est la manifestation physique du génotype, qui est supposé être invisible. Un phénotype peut donc avoir plusieurs génotypes mais l'inverse n'est pas vrai.

Notre premier objectif avec ROBOBO est l'implémentation de comportements de type "danse". Par conséquent nous ne nous servons pas des senseurs du robot et avons donc décidé de modéliser le comportement (ou danse) sous la forme d'une séquence d'instructions basique, il s'agit là du génotype, dont l'exécution produit un comportement, le phénotype.

La mutation

D'après ce que nous avons pu lire dans les différents articles lus et dans Introduction to Evolutionary Computing [1] il est préférable que les probabilités qu'un individu subisse une mutation s'approche de 1. Par conséquent, suivant les conseils trouvés dans le livre [1] nous accordons à chaque gène une probabilité égale à $\frac{1}{len_{séquence}}$ de subir une mutation.

Lors de notre première implémentation chaque gène avait aussi la possibilité de muter vers le gène nul (c'est à dire que le gène est simplement retiré). Le problème avec cette implémentation était que les probabilités étaient trop dépendantes du nombre de gènes différents (chaque type de gène ayant la même probabilité d'être sélectionné) et cela influençait donc la taille moyenne des séquences.

Exemple : Si on a quatre gènes différents et une séquence de longueur 5 on a donc, lors de la mutation, $\frac{1}{6}$ chance de rajouter un gène pouvant être vide, soit en réalité $(\frac{1}{6} \cdot \frac{4}{5} =) \frac{2}{15}$ chance de rajouter un gène à un endroit de la séquence. On a par contre $\frac{1}{5}$ chance d'enlever n'importe quel gène muté, soit $\frac{1}{6} \cdot \frac{1}{5} \cdot 5$ (probabilité de mutation * probabilité de gène nul * taille séquence) $= \frac{1}{6}$ chance de perdre un gène. Avec ces probabilités on arrive donc à une longueur moyenne égale au nombre de gènes différents.

On voulait donc soit décider nous-même de la taille moyenne des séquences, soit faire en sorte que la probabilité de gagner un gène soient égal à la probabilité d'en perdre un. Est-ce que dans le deuxième cas on ne risque pas d'avoir des séquences qui deviennent trop courtes ? Non, ou en tout cas c'est assez improbable (pas au point d'être un problème) car la NS va rapidement éliminer les séquences trop courtes car trop semblables entre elles. Au pire on pourrait démarrer avec la séquence vide en mémoire afin d'assurer qu'elle n'apparaisse pas comme séquence possible.

Après avoir testé les deux méthodes nous avons décidé que les séquences trop longues avaient assez peu d'intérêt pour l'utilisateur (en plus de rallonger nos tests). Nous avons donc décidé d'imposer une longueur de séquence moyenne. De plus, jusqu'à ce point, nous déterminions le gain ou non d'un gène après la mutation de la séquence. Les probabilités de gain et de perte moyennes étaient donc bien calibrées mais pas leur variance. On pouvait, dans les cas les plus extrêmes, d'un côté gagner au plus un gène et de l'autre perdre tous les gènes. Nous avons donc décidé de déterminer la probabilité de gain ou de perte comme dépendante de la probabilité de mutation (et donc comme une forme de mutation possible).

A présent chaque gène a toujours une probabilité de mutation égale à $\frac{1}{len_{séquence}}$ mais cette mutation a 80% de chance de provoquer une mutation classique du gène (c'est à dire qu'il peut devenir n'importe quel autre gène, avec même probabilité pour chaque type de gène) et 20% d'être perdu ou de se voir rajouter un nouveau gène à côté. Dans le deuxième

cas les probabilité de gain ou de perte sont de 50/50 lorsqu'on se situe à la longueur de séquence idéale et ont pour relation $\mathbb{P}(gain) = 1 - \mathbb{P}(loss)$. La probabilité de perte tend vers 1 lorsque la longueur augmente (atteint à deux fois la longueur idéale) et vers 0 lorsque la taille de la séquence diminue (atteint en longueur zéro, soit la séquence nulle).

Le crossover

Après nous être documentés sur plusieurs algorithmes de crossover existants, la plupart expliqués dans Introduction Evolutionary Computing [1], et sommes arrivés à plusieurs conclusions :

- Dans le cas d'un comportement Robobo de type danse une séquence est préférée si elle possède une ou un ensemble de sous-séquences intéressantes. Ces sous-séquences se caractérisent par un ordonnancement particulier de gènes.
- L'algorithme choisi doit être capable de combiner des séquences de longueurs différentes.
- Les proportions de types de gènes doivent être prises en compte.

Nous avons donc opté pour une adaptation d'un Edge-3 crossover d'après Whitley.

Le Edge-3 crossover d'après Whitley tel qu'expliqué dans Introduction Evolutionary Computing [1] : On prend deux séquences de même taille, les parents, chacune composée des même gènes n'apparaissant qu'une fois dans chaque séquence. La seule chose qui différencie donc les deux séquences est l'ordonnancement des gènes. On construit une table d'adjacence des gène. C'est à dire que pour chaque gène on détermine qui sont ses voisins dans les parents, si un voisin est en commun dans les deux parents on le marque comme prioritaire (dans le livre un symbol '+' est ajouté). On choisi aléatoirement un gène de départ. On l'ajoute à la nouvelle séquence et on enlève toute référence à cet élément dans la table d'adjacence. On va chercher l'élément suivant dans la table d'adjacence parmi ses gènes adjacents (chez les parents). On prend en priorité les gènes marqué comme prioritaires. Sinon on prend en priorité l'élément avec la plus petite liste d'adjacence. En cas d'égalité, on tranche au hasard. Si les listes d'adjacence observées sont vides on sélectionne l'élément au hasard parmis les éléments restants.

Notre adaptation : L'algorithme décrit ci-dessus est utilisé pour des listes de même taille et possèdant les mêmes éléments n'apparaissant qu'une fois chacun. Il nous a donc fallu l'adapter afin de pouvoir l'utiliser sur nos séquences. La première étape consiste donc à déterminer quels seront les gènes présents chez l'enfant. On fait donc la liste des gènes présents avec leur nombre total d'occurrences. On détermine la longueur de la séquence enfant en prenant une valeur au hasard entre les longueurs des deux séquences parents. On prend ensuite en priorité le gène ayant le plus grand nombre d'occurrences (on tranche au hasard en cas d'égalité) et on décrémente sa valeur d'occurrence de deux. On continue ainsi jusqu'à avoir atteint le nombre de gènes nécessaire . La deuxième étape est l'ordonnancement des gènes de l'enfant. Comme pour la version d'origine on va donc construire une table d'adjacence pour les éléments présents chez l'enfant. Le reste de l'algorithme est plus ou moins le même, à quelques points près : on prend en priorité les éléments ayant la plus grande valeur d'adjacence (ils ont maintenant une valeur d'adjacence au lieu d'une simple différenciation binaire), les gènes peuvent apparaître plusieurs fois dans la table d'adjacence si ils apparaissent plusieurs fois chez l'enfant mais seule la première ligne rencontrée est ôtée du tableau lorsqu'un élément est ajouté à la nouvelle séquence (dans l'implémentation finale chaque élément ainsi présent dans le tableau possède une valeur d'occurrence qui se voit décrémentée), les valeurs d'adjacence les plus hautes liées au gène

ajouté sont décrémentées (seulement les plus hautes pour éviter de réduire à zéro une autre adjacence potentiellement importante) et si un élément n'est plus présent dans le tableau ses valeurs d'adjacence sont mises à zéro. L'algorithme prend fin lorsque le tableau est vide (toutes les valeurs d'occurrence des lignes à zéro).

Exemple : Soient deux séquences S1 = AAABBC et S2 = CAAC. On calcul les valeurs occurrence : occ(A)=5, occ(B)=2, et occ(C)=3 (soit [5,2,3] On calcul la longueur de l'enfant : len(S1) = 6, len(S2) = 4, donc len(S3) = 5 (au hasard parmi 4,5 ou 6) On détermine les éléments présents : A (nouvelle occ : [3,2,3]), A (occ [1,2,3]), C (occ [1,2,1]), B(occ [1,0,1]), C(occ [1,0,-1]) On construit la table d'adjacence avec ces éléments :

Element	Adjacence
A	A(6),B(1),C(3)
A	A(6),B(1),C(3)
C	A(3),B(1),C(2)
B	A(1),B(2),C(1)
C	A(3) B(1) C(2)

On choisi maintenant un premier élément au hasard (disons C) et on suit l'algorithme

El.	Adjacence
A	A(6),B(1),C(2)
A	A(6),B(1),C(2)
~~C~~	**A(3)**,B(1),C(2)
B	A(1),B(2),C(1)
C	A(3) B(1) C(2)

El.	Adjacence
~~A~~	**A(5)**,B(1),C(2)
A	A(5),B(1),C(2)
~~C~~	~~A(3),B(1),C(2)~~
B	A(1),B(2),C(1)
C	A(3) B(1) C(2)

El.	Adjacence
~~A~~	~~A(5),B(1),C(2)~~
~~A~~	~~A(4)~~,B(1),**C(2)**
~~C~~	~~A(3),B(1),C(2)~~
B	~~A(1)~~,B(2),C(1)
C	~~A(3)~~,B(1),C(2)

El.	Adjacence
~~A~~	~~A(5),B(1),C(2)~~
~~A~~	~~A(4),B(1),C(2)~~
~~C~~	~~A(3),B(1),C(2)~~
B	~~A(1)~~,B(2),~~C(1)~~
~~C~~	~~A(3)~~,**B(1)**,~~C(1)~~

On obtient donc le résultat suivant : **S3 = CAACB**

Alternative testée : Au début nous avions aussi testé un simple One-Point crossover qui donnait des résultat bien moins intéressants. L'algorithme consistait à prendre aléatoirement un point i d'une valeur comprise entre 0 et la longueur de la plus petite séquence (parmi les deux parents, bornes excluses). Ensuite on construit l'enfant en prenant le début d'une des deux séquences jusqu'au point i, puis la suite de l'autre à partir du point i (à noter qu'il est alors préférable de créer deux séquences enfant si on ne veut pas perdre d'information des parents).

Le Filtrage NS

Le principe de la Novelty Search [2] est de générer des nouvelles générations peuplées d'individus qui résultent en des comportements suffisamment différents des comportements obtenus jusqu'alors (suffisamment nouveau). Il faut donc garder en mémoire une description des comportements suffisante pour pouvoir effectuer la comparaison avec les nouveaux comportements. Attention cette comparaison est faite sur les comportements, on est donc au niveau phénotypique.

Il nous a donc fallu choisir un moyen de comparaison des comportements et déterminer comment calculer un taux de nouveauté acceptable. Pour la comparaison des comportements, notre choix de codage du génotype étant suffisamment proche du comportement résultant, nous avons utilisé un simple calcul de distance d'édition de Levenshtein. L'autre option est la comparaison des images obtenues par simulation du comportement. Cependant, dans nos tests la première méthode s'est jusqu'à maintenant avérée suffisante. Pour déterminer la distance minimum requise pour pouvoir faire partie de la nouvelle génération nous comparons les distances des parents sélectionnés et calculons une valeur de coupe à

partir de ces valeur. Le calcul de cette valeur a déjà été maintes fois modifié : la plus petite valeur parmi les parents sélectionnés (par rapport à tous les autres comportement), juste la première valeur ainsi retournée, la moyenne des distance maximum des parents entre-eux, le maximum des distances minimum, etc. Actuellement cette valeur est le minimum de la moitié arrondie au supérieur (ou 1 si 1 est supérieur) des maximums des valeurs de nouveauté des parents entre eux, cette valeur sera sans nulle doute modifiée au moins une fois de plus après l'écriture de ce chapitre.

Lors de la création de la nouvelle génération, les nouveaux individus sont générés les uns après les autres jusqu'à obtention d'un nombre suffisant d'individus, ou jusqu'à écoulement du temps maximum alloué à l'algorithme. Lorsqu'un nouvel individu est généré, s'il est suffisamment différent des autres (c'est à dire que son score de nouveauté est supérieur ou égale à la valeur de coupe) il est ajouté à la nouvelle génération et à la liste des individus précédents (pour les prochains calculs de nouveauté), sinon il est simplement rejeté.

La simulation

La simulation ne fait pas partie à proprement parler de l'algorithme évolutionnaire. Elle est pourtant utilisée pour permettte à l'utilisateur de visualiser les comportements, afin de les selectionner pour la génération de la population suivante. Nous avions prévu d'utiliser FastSim qui est un module de Sferes2. Malheureusement, nous n'avons, pas réussi à integrer du code C++ à notre projet. Nous avons donc mis au point un simulateur basique permettant de décrire les mouvements de ROBOBO.

Nous avons dû passer par une phase de test afin de calibrer le simulateur pour qu'il corresponde le plus possible à ROBOBO, et qu'en même la trajectoire soit la plus lisible pour l'utilisateur. Sur la visualisation des trajectoires, les points rouges désignent le début du comportement, les points jaunes le milieu, et les points bleus la fin.

5.2 Implémentation et Complexité

RobobGene

- **Mutate** : Cette méthode fait muter un gène. On prend un mouvement au hasard parmi les RoboboMvmt et on l'attribue à ce gène. Ainsi, un gène qui mute a une faible chance de ne pas changer. Elle se finie en temps constant.

RoboboDNA

- **Mutate** : Fait muter un génotype ADN. Pour un ADN de longueur n, cette méthode a une complexité en $O(n)$.
  On crée une liste de gène vide $newGenotype$, et on pose $lossProba = \min(1, \frac{currSize}{2*idealSize})$.
  Pour chaque gène G de ADN :
  On tire un flottant r entre 0 et 1.
  Si $r >= 0.2$ on fait muter G et l'ajoute à $newGenotype$.
  Sinon, on tire r' entre 0 et 1.
  Si $r' >= lossProba$, on génére un nouveau gène totalement aléatoire et l'ajoute à $newGenotype$.
  Finalement, on retourne $newGenotype$.

RoboboPopulation

- **NoveltySearch** : Pour toute la population on regarde la valeur de la plus petite distance de Levenstein. Celle-ci sera la distance minimale pour accepter un enfant généré par le crossover. Puis on affiche les enfants acceptés.

- **xOver** ; Pour deux parents de taille n et m, le crossover se fait en $O(n+m)$.
  CROSSOVER algo

```
class occ_obj

    gene_type gene
    int occ_val

class occ_list

    array<occ_obj>

    get_val(gene_type) :
        return occ_val or 0 (if not in list)

    get_all(int value) :
        return list of genes with occ_val == value

    sort() :
        sort by greatest occ_val first

    add1(gene_type) :
        if in list add 1 to value, else add to list with value 1

    sub1(gene_type) :
        substract 1 to value of gene_type

    sub2(gene_type) :
        substract 2 to value of gene_type

crossover (parent_list) :

    randomGenerator = new Random();
    parent_A = parent_list.get(randomGenerator.nextInt(parent_list.size()));
    parent_B = parent_list.get(randomGenerator.nextInt(parent_list.size()));

    if parent_A == parent_B (same pointer)
        return copy(parent_A)

    min_size = min(len(parent_A), len(parent_B))
    max_size = max(len(parent_A), len(parent_B))

    if max_size == minsize
        child_size = min_size
    else
```

```
    child_size = randomGenerator.nextInt(max_size + 1 - min_size) + min_size

gene_list = new occ_list
child_gene_list = array<gene_type>
occurence_tab = new array<occ_list>

for p_gene in parent_A + parent_B
    gene_list.add1(p_gene)

gene_list.sort()

for (i=0, i < child_size, i++)
    candidates = gene_list.get_all(gene_list.get(0).occ_val)
    chosen = select random candidate
    child_gene_list.add1(chosen)
    gene_list.sub2(chosen)
    gene_list.sort()

for gene in child_gene_list
    occurence_line = new occ_list
    // for parent_A
    for (i=0, i < len(parent_A), i++)
        if parent_A[i] in child_gene_list
            if parent_A[(i-1)%parent_A.size()] in child_gene_list
                occurence_line.add1( parent_A[(i-1)%parent_A.size()] )
            if parent_A[(i+1)%parent_A.size()] in child_gene_list
                occurence_line.add1( parent_A[(i+1)%parent_A.size()] )
    // same for parent_B

    occurence_line.sort()
    occurence_tab.add(occurence_line)

best_occ_list = make occ_list using maximum values in occurence_tab

child = new array<gene_type>
select random starting point i in gene_child_list

while (!child_gene_list.isEmpty())
    //first consequences of choice
    added_gene = child_gene_list[i]
    child.add(added_gene)
    if(child_gene_list.size() > 1)
        for occ_line in occurence_tab
            if occ_line.get_val(added_gene) == best_occ_list.get_val(added_gene)
                occ_line.sub1(added_gene)
                occ_line.sort()
        best_occ_list.sub1(added_gene)
    child_gene_list.remove(i)
    occurence_tab.remove(i)
```

```
        //next choice of i
        if(!child_gene_list.isEmpty())
            candidates = occurence_tab[i].get_all(occurence_tab[i].get(0).occ_val)
            remove from candidates all not in child_gene_list
            if candidates == null
                i = randomGenerator.nextInt(child_gene_list.size())
            else
                gene_type next = new pick random from occurence_tab[i].get_all(occure
                i = index of first occurence of 'next' in child_gene_list

    return child
```

- **distLevenshtein** : Retourne la distance de Levenshtein (ou encore distance d'édition) entre un individu 1 (dont le génotype est de longueur n) et un individu 2 (de taille m). La complexité de cet algorithme est en $O(n * m)$, et utilise le principe de la programmation dynamique.

- **minLevenstein** : Retourne la plus petite distance qui sera accepté entre la nouvelle génération et les parents.

- **levensteinCutoff** : Retourne la distance qui servira de seuil pour le Novelty Search.

- **choisirParent** : Retourne un parent parmi la population. Plus un parent est selectionné, plus la probabilité qu'il soit à nouveau selectionné diminue. Si la population est de taille n, la complexité de cette fonction est en $O(n)$.

RoboboDNASim

Le ROBOBO simulé est caracterisé par la donnée d'un vecteur position, d'un vecteur direction et d'une vitesse. La position est initialisée à (100,100),la direction à (0,-1) et la vitesse à 0. Ensuite, chaque mouvement de chaque gène de l'ADN considéré est tranformé en action, c'est-à-dire un couple (accélération, angle).
La nouvelle direction est obtenue par rotation du vecteur direction N de l'angle desiré. La vitesse est calculée grâce à une formule simple prenant en compte des frottements liées à ROBOBO lui même et le milieu sur lequel il se déplace. Dans la pratique, lors du calibrage du simulateur, les resultats était satisfaisant lorsque ces coefficients étaient faibles ($\simeq 10^{-3}$).
La vitesse du ROBOBO étant physiquement limitée, nous avons determiner arbitrairement un Vit_{max} qui ne sera jamais depassée.
Finalement, la nouvelle position est calculée à partir de la position précédente, et des nouvelles direction et vitesse.
Par exemple, dans la simulation en est au pas N, et reçoit l'action (α, θ). On a alors :
$Dir_{N+1} = \mathcal{R}(\theta).Dir_N$, où $\mathcal{R}(\theta)$ est la matrice de rotation d'angle θ.
$Vit_{N+1} = \min(Vit_{max}, \alpha \times a - Vit_N \times b - c)$ où a, b et c sont les constantes liées au modèle déplacement.
$Pos_{N+1} = Pos_N + Dir_{N+1}.Vit_{N+1}$
Lors de la simulation la plupart des déplacements impliquant de l'acceleration sont effectués plusieurs fois par le simulateur, pour les rendre bien visible sur le rendu final qui sera donné à l'utilisateur.

6 Résultats des tests

Première Génération

Comportements sélectionnés

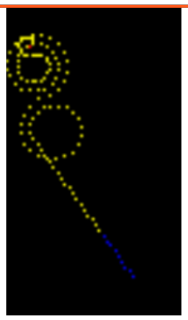


Figure 1


Figure 2


Figure 3

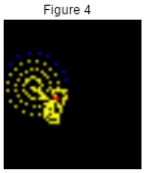


Figure 4

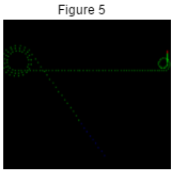


Figure 5

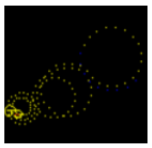


Figure 6

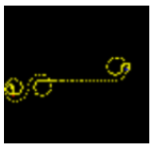


Figure 7

Deuxième Génération

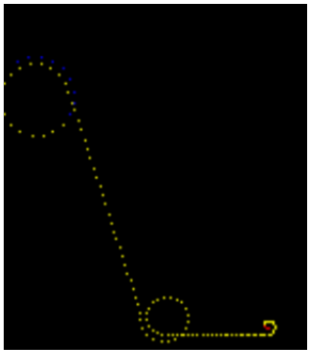


Figure 8

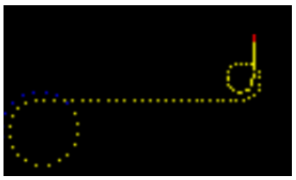


Figure 9


Figure 10


Figure 13

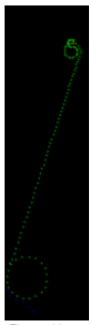


Figure 11

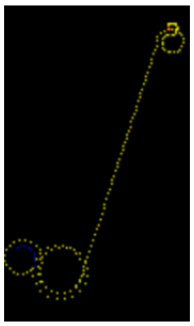


Figure 12

Troisième Génération

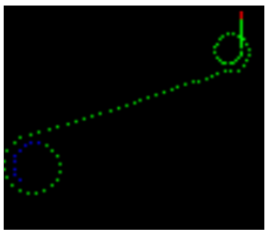


Figure 14

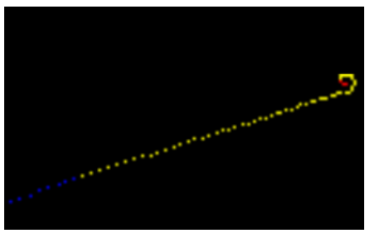


Figure 15

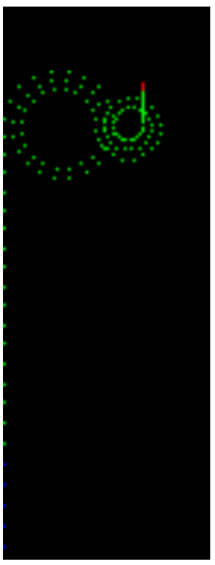


Figure 16

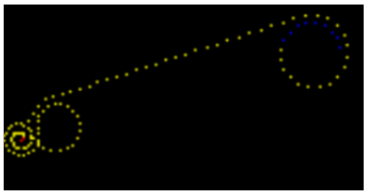


Figure 17


Figure 18

Les images ci-dessus représentent les comportements générés à chaque génération. On peut constater que certain gènes ont bien été gardé par l'algorithme lors de l'utlisation de l'application. De plus certain comportements non sélectionnés lors de la 1er génération ne sont pas proposés à la génération 2 et 3.

De la Génération 1 à 2

Les comportements de la génération 2 commencent par des déplacements circulaires puis se déplacent tout droit, et termine en tournant sur eux-même, comme les comportements 2 et 3 de la génération antérieure.
Par ailleurs la fin du comportement 1 ressemble à la fin de la figure 13, mais l'angle n'est pas le même.

De la Génération 2 à 3

Après avoir sélectionné la figure 8 on remarque que certains comportements vont faire avancer le robot avec un angle different, voir figure 14,15,17.
De plus, on observe que le début de la figure 9 (les pointillés rouges) est similaire au début des figures 14 et 16. Aussi la fin du comportement de la figure 13 ressemble à la fin de la figure 15 et 16.

7 Conclusion

Dans le cahier des charges, nous parlions d'utiliser des réseaux neuronaux, mais par manque de temps nous n'avons pas pu nous y interesser.
Bien que nous soyions satisfaits des résultats de notre travail, nous savons qu'il est encore possible de l'améliorer. Par exemple, nous avions prévu d'utiliser une coupe de Levenshtein dans notre implémentation du Novelty Search afin de l'améliorer. De plus, les comportements que nous avons modéliser ont des paramètres fixes. En effet, les valeurs par défaut de vitesse et de temps d'exécution sont définie au début, et ne sont jamais modifiées. Ceci a pour avantage de rendre la simulation et la modélisation plus simple, mais limite quelque peu l'originalité des danses.
Nous avons été prévenu que l'utilisation du robot dans les tests pratiques pouvait amener des problème supplémentaires. Nous y avons confrontés plusieurs fois. Par exemple, le robot se deconnecte régulierement de l'application sans raisons apparentes. Ensuite, certains modes par défaut de robot l'empecher de se déplacer sur certaines surfaces.
Ce projet nous a permis d'étendre nos connaissances à la fois sur la robotique et sur les algorithmes évolutionnistes. Nous avons pu travaillé sur de vrais robots, et avons realisé leur limite. Nous sommes maintenant bien conscients de l'étendue des forces et faiblesses des algorithmes génétiques.

8 Bibliographie

Références

[1] Agoston E. Eiben et J. E. Smith. Introduction to Evolutionnary Computing. 2003. isbn : 9783662050941.

[2] Brian G. Wooley et Kenneth O. Stanley. "A Novel Human-Computing Collaboration : Combining Novelty Search with Interactive Evolution". In : (2014). In Proceedings of the 16th annual conference on Genetic and evolutionary computation.