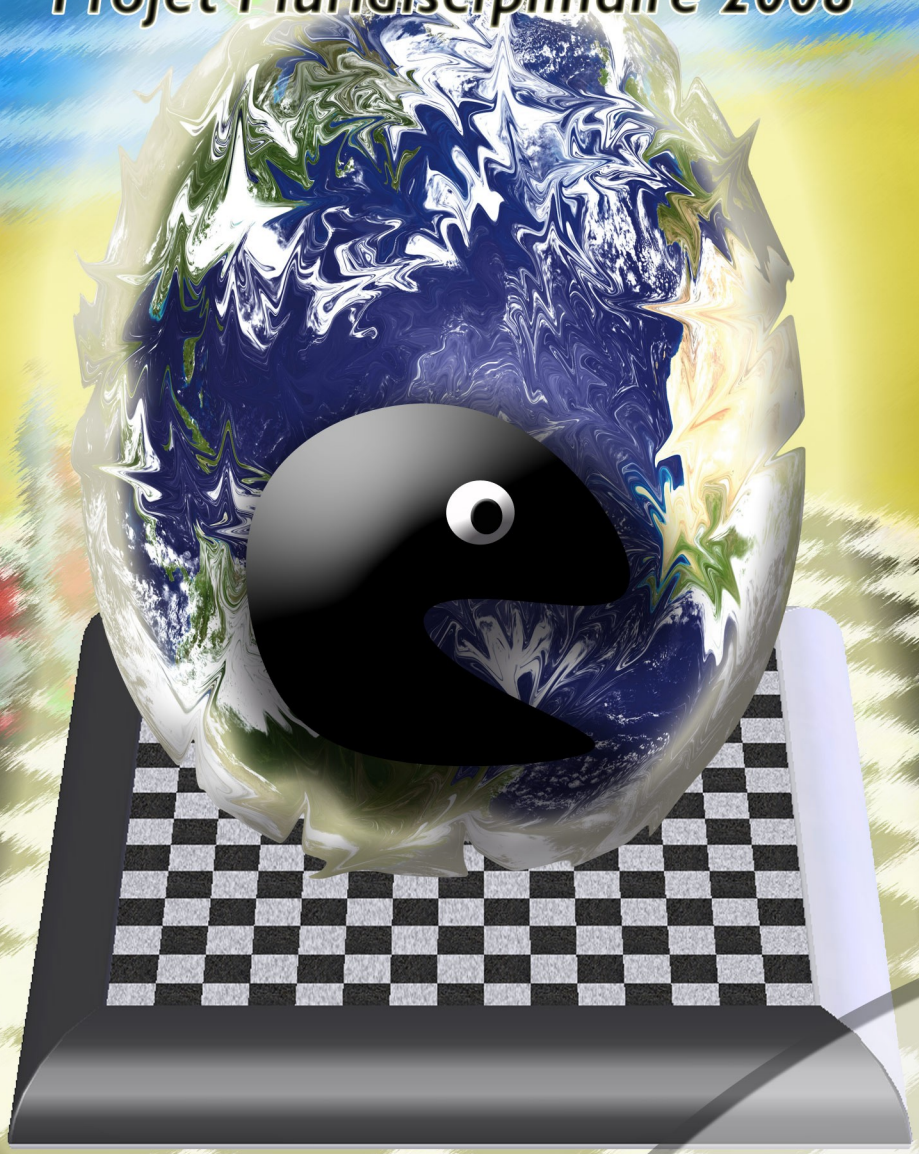
esiea
paris

# Utilisation des méthodes évolutionnistes dans la découverte de stratégies

## Projet Pluridisciplinaire 2008

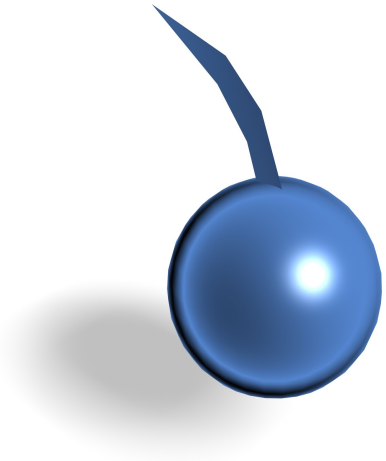



Guillaume CHARTON
Michaël DARMES
David LEBLANC
Laurent MORILLON

Suiveur du projet : M. David JUILIEN

# Utilisation des méthodes évolutionnistes dans la découverte de stratégies

***Objet du projet***
*L'implémentation d'un algorithme évolutionniste dans un jeu de simulation d'êtres vivants*



**Thématique :** Intelligence artificielle et recherche de stratégies
**Disciplines mise en oeuvres :** Algorithmique, mathématiques et programmation

**Auteurs :**
- Guillaume CHARTON (Classe 21) <charton@et.esiea.fr>
- Michaël DARMES (Classe 22) <darmes@et.esiea.fr>
- David LEBLANC (Classe 21) <leblanc@et.esiea.fr>
- Laurent MORILLON (Classe 21) <morillon@et.esiea.fr>

**Remerciements à** :
Tahir Vico, avec qui nous avons conçu l'animation 3D pour rendre plus concret ce projet abstrait.

**Suiveur du projet :** Monsieur JULIEN David

# Sommaire

# Introduction

*« L'intelligence artificielle est formée d'algorithmes qui, implantés sur un ordinateur, permettent à ce dernier de réaliser des travaux, qui s'ils étaient réalisés par des hommes, seraient qualifiés d'intelligents. »*[1]

Vous êtes-vous déjà demandé s'il était possible de reproduire l'évolution de l'humanité avec un simple programme informatique ? S'il n'y avait pas une logique dans notre mode de fonctionnement et d'adaptation à l'environnement ? Des chercheurs se sont penchés sur le problème pendant des siècles pour aboutir à la conclusion qu'effectivement, code génétique et code informatique pouvaient devenir bons amis.

Les algorithmes évolutionnistes sont de ceux-là ; ils confrontent une espèce à un problème donné et la font évoluer de façon à ce qu'elle devienne apte à le résoudre. Pour se faire, il y a génération aléatoire d'une population initiale qui va être confrontée au problème et notée sur sa performance. Le résultat fourni est une succession de sélections des meilleurs espèces qui abouti à celle qui aura obtenu le meilleur score.

Les idées de Darwin nourrissent énormément celles des algorithmes évolutionnistes puisqu'elles se fondent majoritairement sur leurs théories ; Darwin avait défini trois règles d'existences :

*« Les individus sont différents les uns des autres »*
*« la nature ne suffisant pas à nourrir tous les individus, cela implique une lutte des espèces pour leur survie »*
*« les individus les plus aptes à survivre ont plus de chance de transmettre leurs caractères avantageux à leur descendance »*

Sur ces idées fut créé *Darwin's World,* un jeu de simulation d'un monde « vivant » qui met en compétition deux espèces de créatures. Le but de chaque espèce est de rallier les créatures ennemies à son camp.
Pour cela, chaque créature a un nombre d'actions limité par tour : *Avancer, tourner sur elle même et infecter*. Les actions que la créature choisit de faire dans chaque situation définissent son code génétique. Chacune des deux espèces qui entre en jeu est régit par son propre code génétique.

Nous avons repris les règles de ce jeu en considérant l'une des espèces comme l'ennemie dont le comportement ne change pas, et une liste d'espèces dont le code est généré aléatoirement qui correspond à la population initiale des algorithmes évolutionnistes.
Nous avons décidé d'évaluer les créatures en récupérant le nombre de points accumulé par chacune d'elles au cours d'un affrontement contre l'ennemie dans *Darwin's World*.

L'objectif du projet était donc d'implémenter le code de mutation en langage C et d'étudier le graphique d'évolution des populations. Nous avons aussi décidé d'écrire un petit logiciel qui nous permettrait de visionner l'une des parties jouées par les créatures.

Nous commencerons ce rapport en expliquant plus en détail ce que sont les algorithmes évolutionnistes. Nous continuerons par une description du jeu *Darwin's World* pour arriver aux objectifs que nous nous étions fixés pour ce projet. Nous terminerons notre développement par une étude des résultats obtenus grâce au programme dont nous sommes les auteurs.

1 Igor ALEKSANDER, introduction à la conception des systèmes intelligents, Ed. HERMES, 1985, Ch. 4 p75

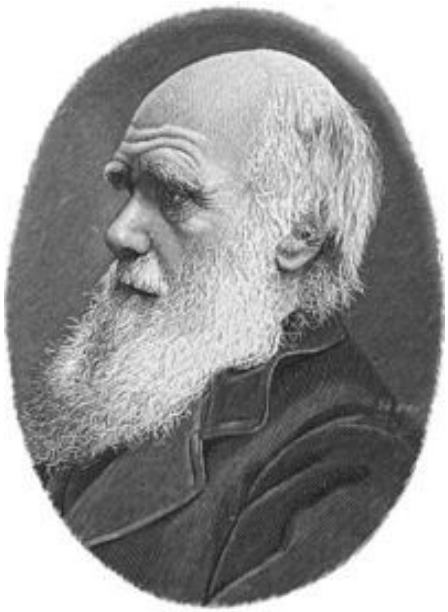

esiea
groupe

# 1. Les algorithmes évolutionnistes

## 1.1. L'inspiration darwinienne

Lorsque les méthodes algorithmiques classiques ne permettent pas de résoudre efficacement les problèmes, on a recourt à des méthodes dites « évolutives » qui s'inspirent du comportement du vivant, ainsi que des principes suivants énoncés par la théorie de Darwin sur l'évolution :

- les individus sont différents les uns des autres
- la nature ne suffisant pas à nourrir tous les individus, cela implique une lutte des espèces pour leur survie
- les individus les plus aptes à survivre ont plus de chance de transmettre leurs caractères avantageux à leur descendance



*Charles Darwin* [3]

Les concepts algorithmiques qui décrivent les méthodes évolutives descendent des observations de Darwin. Il existe deux types d'algorithmes s'inscrivant dans les méthodes évolutives :

- les algorithmes génétiques dont le principe est de croiser les caractères des individus les plus forts
- les algorithmes évolutionnistes dont le principe de faire muter les caractères des créatures en reprenant entièrement les caractères des individus les plus forts

Dans le cadre du PPD, nous nous pencherons uniquement sur les algorithmes évolutionnistes.

## 1.2. Explications

Les algorithmes évolutionnistes consistent à générer aléatoirement une population initiale composée d'individus totalement indépendants les uns des autres. On attribue à chaque espèce générée, une note appelée fitness définie sur des critères de comportement face à un problème.
Une fois cette population notée, elle est triée selon le fitness.

Deux solutions se présentent à nous quant à l'évolution :

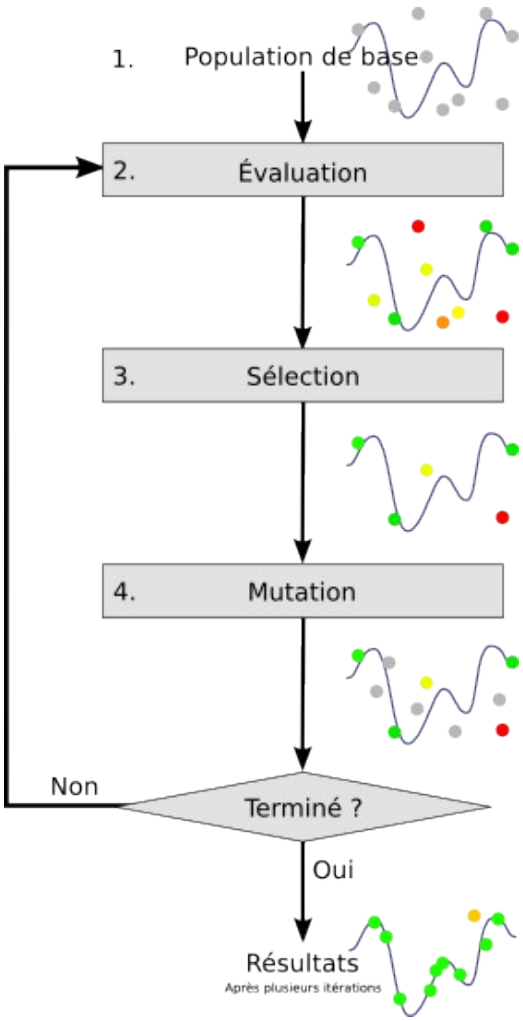
- Le croisement des espèces qui consiste à sélectionner deux espèces de la liste et à mélanger leur code pour en obtenir une nouvelle.
- La mutation des espèces qui consiste à modifier aléatoirement une partie de leur code pour en obtenir une nouvelle.

Cette opération est répétée autant de fois qu'on le souhaite pour aboutir à une liste d'espèces contenant la meilleur solution pour remédier au problème de départ.

## 1.3. Historique

- 1860 : Charles Darwin publie son livre intitulé *L'origine des espèces* au moyen de la sélection naturelle ou la lutte pour l'existence dans la nature.
- 1954 : Barricelli effectue les premières simulations du processus d'évolution et les utilise sur des problèmes d'optimisation généraux.
- 1956 : Workshop où est né le terme « intelligence artificielle »
- 1960 : John McCarthy, Allen Bewell & Herbert Simon: L'ordinateur peut être utilisé pour autre chose que des calculs « manipuler des symboles » (idée proposée par Ada Lovelage amie de Babbage 1842)
- 1962 : John Holland met en place la base des Algorithmes génétiques binaires.
- 1965 : Rechenberg conçoit le premier algorithme utilisant des stratégies d'évolution.
- 1966 : Fogel, Owens et Walsh proposent la programmation évolutionnaire.
- 1970 : John Horton Conway conçoit *le jeu de la vie*, l'automate cellulaire le plus connu à ce jour.
- 1973 : Stratégie d'évolution I. Rechenberg
- 1975 : Travaillant sur les automates cellulaires, Holland propose les premiers algorithmes génétiques.
- 1988 : La première conférence sur les algorithmes génétiques est organisée à l'université de l'Illinois à Urbana-Champaign.
- 1988 : Des travaux sur le comportement collectif des fourmis trouvent une application en intelligence artificielle.
- 1989 : Evolver, le premier logiciel d'optimisation par algorithmes génétiques est publié par la société Axcelis.
- 1993 : Le terme *Evolutionary Computation* (calcul évolutionnaire en français) se répand, avec la parution de la revue éponyme, publiée par le Massachusetts Institute of Technology.
- 1997 : Storn et Price proposent un algorithme à évolution différentielle.
- 2000 : Premiers algorithmes génétiques interactifs.

## *1.4. Schéma fonctionnel*




1. Générer aléatoirement une population initiale.

2. On confronte cette population au problème. Dans notre cas, le problème est d'éliminer l'adversaire. On fait donc jouer cette population.

   A la fin du combat, on évalue l'efficacité (fitness) de chacun des individus de cette population.

3. On favorise le choix d'un ou des individus les plus forts comme base pour la future génération (sélection naturelle).

4. On modifie un ou plusieurs caractères de la sélection au hasard formant ainsi la nouvelle population.

5. On évalue si le nombre de mutations est assez grand pour obtenir une population qui possède une stratégie suffisamment efficace pour résoudre le problème.

   Dans le cas contraire, on réitère l'algorithme à partir de l'étape 2.

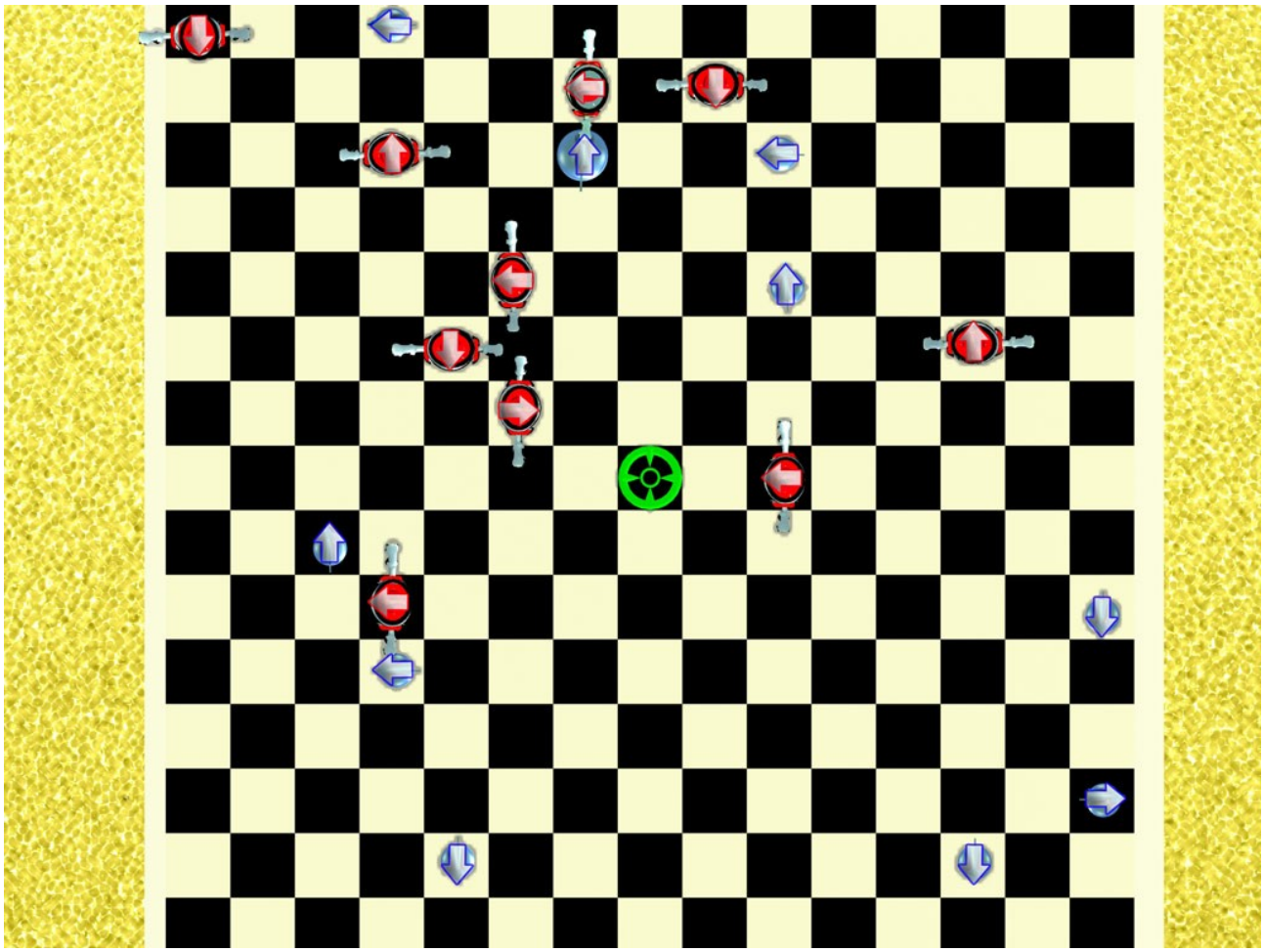
Dans un premier temps, le projet consistera, par le biais de la méthode évolutionniste, à observer l'évolution d'un comportement passif au premier abord en un comportement apte à résoudre un problème particulier.

Nous optimiserons par la suite afin d'obtenir une évaluation de fitness et les mutations les plus fines possibles.

## 2. Les règles du jeu Darwin's World

*Il est important de présenter le jeu sur lequel vont s'appuyer nos recherches :*

*Darwin's World de Nick Parlante <http://nifty.stanford.edu/darwin/>.*



Ce jeu est un combat entre deux espèces de créatures. Chaque espèce est régie par un programme informatique simple écrit dans un pseudo-langage. On dispose aléatoirement un nombre défini de créatures sur un damier ; les deux équipes ayant le même nombre de créatures. Durant un tour de jeu, chaque entité peut exécuter quatre actions différentes :

- tourner à droite
- tourner à gauche
- avancer sur la case en face
- infecter la case en face

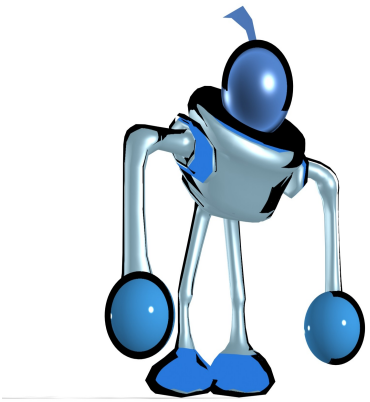
L'infection est utile seulement si la case en face de la créature contient un ennemi. Dès lors, celui-ci deviendra une créature de la même espèce que celle qui l'a infecté. Cette action est considérée nulle dans toute autre situation mais compte comme un tour. Un tour de jeu se termine lorsque l'ensemble des créatures du damier a joué une fois.

Il faut aussi ajouter que chaque créature peut vérifier si la case qui lui fait face contient un ennemi, un ami ou un mur. Pour gagner, il faut éliminer la population adverse en l'infectant.

# 3. Objectifs de notre projet

## 3.1. Cahier des charges

Au début du projet, nous avons été confronté à une grande période de recherche pour comprendre la logique des algorithmes évolutionnistes et les théories de Darwin. Nous avons ensuite pu concevoir un cahier des charges où nous définissions notre propre algorithme évolutionniste.



Notre objectif était tout d'abord de produire un logiciel de simulation ; ce logiciel aurait comme but de faire évoluer des créatures au cours d'un nombre défini de parties. Le but de cette évolution est de vaincre un ennemi en ralliant les créatures de son espèce à celle générée par le logiciel.

L'objectif principal était d'aboutir à deux programmes :

- Un programme de traitement qui génère les espèces pour les tester contre un ennemi au cours d'une partie de Darwin's World.
- Une interface graphique permettant de visualiser le jeu sous forme d'une animation 2D vue de dessus.

Nous avons utilisé le langage C pour le logiciel de simulation, et le langage Delphi pour l'interface graphique qui lance une animation à partir du script généré par le logiciel de simulation.

La deuxième partie de notre objectif consistait en l'analyse des résultats produits par notre logiciel et la recherche des paramètres les plus aptes à nous offrir une évolution efficace.

Pour cela, nous avons intégré une liste de constantes modifiables à notre programme pour pouvoir lancer plusieurs simulations avec des paramètres différents.

Les constantes sont présentées de la manière suivante :

```
/*valeurs à changer dans l'étude*/
#define Tour_per_party 10000 /*Nombre de tour par partie*/

#define PuissanceDix_tour_per_party 1000 /* par exemple si Tour_per_party=6000=6*10^3
donc PuissanceDix_tour_per_party=10^3=1000*/

#define Nbr_party 1 /*nombre de parties à jouer avant d'afficher la meilleur creature*/

#define party_per_moyenne 1 /*nombre de parties sur lesquelles évaluer une seule espèce*/
#define coeff_fitness_health 1 /*coefficient appliqué aux points de vie*/
#define coeff_fitness_infect 5 /*coefficient appliqué aux points d'infection*/
#define Nbr_change_lines 1 /*nombre de lignes à changer à chaque évolution d'espece*/
#define affiche_especes_similaires 0 /*Affiche la liste des espèces générées qui ont
obtenu les mêmes résultats*/
#define affiche_liste_especes 1 /*affiche la liste des espèces et demande si on veut
afficher le code de l'une d'entre elles*/
```

## *3.2. Notre planning prévisionnel*

**26 Octobre 2007**

Rédaction du cahier des charges

**Décembre 2007**

Recherche de procédés d'évaluation de fitness et de mutation.
Programmation : Conception de la partie simulation du logiciel
(recherche des algorithmes)
Finalisation du cahier des charges qui servira de support

**Janvier 2008**

Implémentation en C des procédés trouvés et imaginés par nous
Programmation : Conception de la partie traitement du logiciel
(récupération et exploitation des données des simulations, générateurs de hasard)

**Février 2008**

Finalisation du logiciel d'évolution
Ecriture du logiciel d'animation

**Mars 2008**

Étude des êtres évolués issus du logiciel
Présentation publique du projet

# 4. Les termes utilisés

De façon à ce que tout le monde comprenne le langage que nous avons adopté au cours du projet, nous avons choisi de décrire notre vocubulaire.

## *4.1. Le code d'une créature*

Nous avons défini trois types d'instructions :

*Les instructions conditionnelles*

ifenemy N
: Si un ennemi se trouve devant, on exécute la ligne N du code de la créature.

ifempty N
: Si la case devant est vide, on exécute la ligne N du code.

ifwall N
: S'il y a un mur devant, on exécute la ligne N.

ifsame N
: S'il y a un ami devant, on exécute la ligne N.

ifrandom N
: Utilise une autre instruction conditionnelle sélectionnée au hasard. Si celle-ci est vérifiée, il exécute la ligne N.

*Les instructions de saut de code*

go N
: Exécuter l'instruction de la ligne N

*Les instructions d'action*

left
: Tourner vers la gauche

right
: Tourner vers la droite

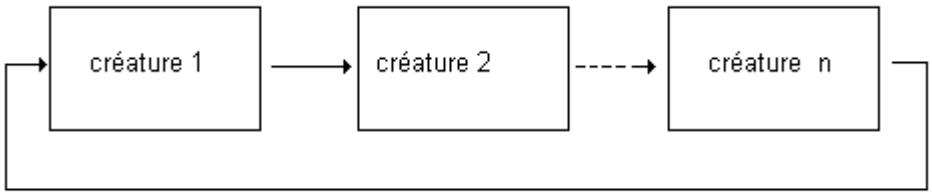
infect
: Infecter l'ennemi

hop
: Avancer si la case de devant est vide ou qu'il n'y a pas de mur

## 4.2. vocabulaire du projet

***Liste exécutive :*** *Un tour de jeu signifie que chacune des créatures des deux équipes a joué. Pour savoir dans quel ordre les faire jouer, nous devons créer une liste dont chaque élément correspond à une créature. Cette liste d'abord triée (liste équipe 1 suivie de liste équipe 2) est mélangée ensuite.*

*On associe ce mélange au hasard de la vie.*



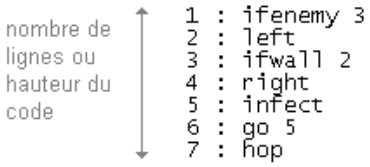

*Informations d'une cellule de la liste exécutive :*
- *coordonnées x,y*
- *équipe*
- *pointeur de code*

***Note :*** *Nombre de points accumulés par une créature pendant une partie.*

***Fitness :*** *Dans le vocabulaire des algorithmes génétiques, le fitness est une note qui est attribuée à chaque espèce de créatures qui a été confrontée au problème. C'est grâce à cette note que l'on peut déterminer qu'une créature est plus forte qu'une autre. On définit cette note entre [0;1[. L'objectif est donc d'augmenter au maximum la valeur de cette note. La traduction anglaise du mot est « adaptation ».*

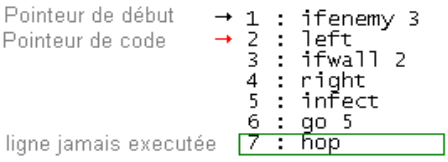
***Créature :*** *Une créature est définie par sa localisation sur le damier, son appartenance à une espèce, son pointeur code et son propre fitness.*

***Code-créature :*** *Chaque créature agit en fonction d'un code. Ce code est composé d'un nombre de lignes défini à l'avance. Chaque ligne contient l'un des trois types d'instructions que nous avons défini dans le I.1 (partie précédente).*



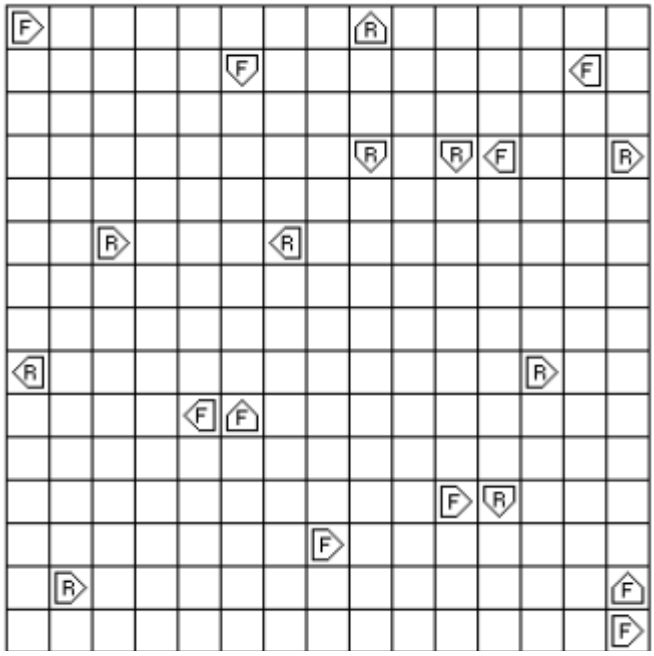

***Ligne-créature exécutée :*** *C'est la ligne de code de la créature qui est exécutée par le programme de simulation.*

***Pointeur de code :*** *C'est la ligne de code de la créature à laquelle le simulateur s'est arrêté au tour précédent. Lorsqu'une créature joue, son code s'exécute de haut en bas jusqu'à arriver sur une instruction d'action. Au tour suivant, le code de la créature continue d'être lu à partir de la ligne qui suit cette instruction.*



***Le damier :*** *C'est le tableau dans lequel se trouve chaque créature.*

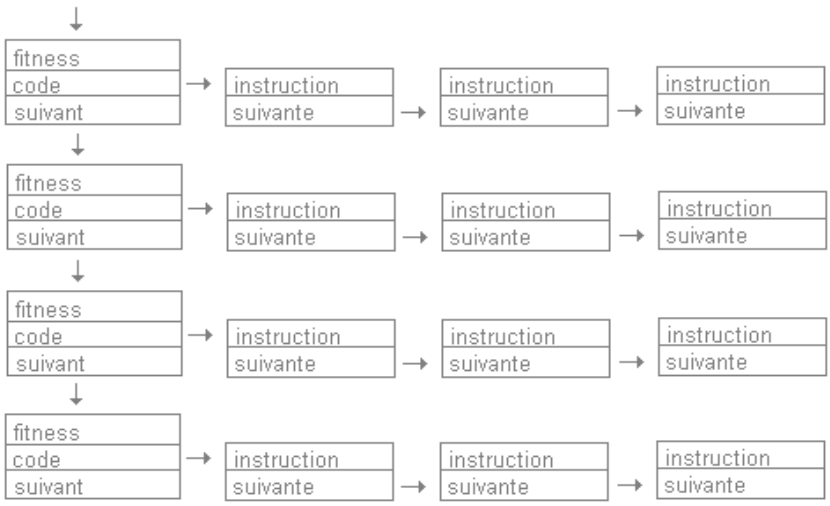
F = créature amie
R = créature ennemie



*Elle permet à la créature de vérifier la présence éventuelle d'ennemis, de murs, de savoir si elle peut se déplacer sur la case voulue (Est-elle vide ? / Fait-elle partie du monde ?)*

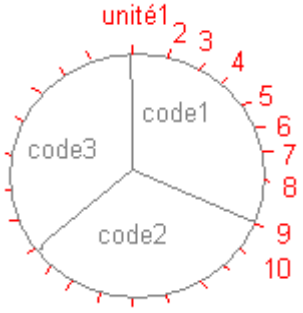
*On définit donc cette structure par un tableau (tab[15][15] permettant un repérage simple avec coordonnées (x,y)) dont chaque case « habitée » pointe vers une créature de la liste exécutive (Elle vaut NULL si la case est vide).*

***Liste des fitness*** *: C'est la liste de tous les codes générés par le simulateur accompagnés de leur fitness.*




*C'est de cette manière que nous gardons les algorithmes en mémoire.*

***Le camembert*** *: C'est un ensemble de N unités distribuées aux codes générés en fonction de leur fitness. Chaque unité a une place P. On tire au sort la place a sélectionner. Le code qui s'est vu attribué l'unité de place P est celui qui va être utilisé pour la génération suivante.*




*Exemple : Si P=8, sur ce camembert, c'est le code 1 qui est sélectionné.*
*Si P=10, c'est le code 2 qui est sélectionné.*

# 5. Notre projet

## *5.1. Développement*

### 5.1.1. Programme de simulation

Dans un premier temps, nous nous sommes concertés pour trouver la façon dont nous allions représenter les différents éléments du jeu et les données que nous voulions garder.

Nous avons opté pour l'organisation en structures comme indiqué dans le sujet du projet « Darwin's World » de l'université de Stanford. Les sources du programme sont agencées en plusieurs fichiers dans le but de rendre le développement plus accessible.

#### 5.1.1.1. Entrées et sorties

Les constantes de la simulation sont déclarées dans le fichier entête *darwin.h*.

Le programme peut également afficher les détails d'une partie, récupérables via une redirection de la sortie standard et exploitables par le programme d'animations.

#### 5.1.1.2. Schémas fonctionnels

##### 5.1.1.2.1. Créature

Si la créature n'a jamais joué (au début de la partie), son pointeur de code pointe sur la première ligne (Initialisation du début de partie).

Le code se lit à partir du pointeur de code jusqu'à tomber sur une instruction d'action ; Donc elle peut lire autant de vérifications que son code lui permet en un tour.

- si le nombre de lignes analysées dans le même tour est supérieur au nombre de lignes du code d'une créature, il y a des chances qu'on se trouve dans une boule infinie : dans ce cas, renvoyer un fitness nul (NOFITNESS) !

- sinon, exécuter l'instruction d'action et modifier le pointeur de code pour qu'il pointe sur la ligne suivante à exécuter. (ligne lue +1 si elle existe, sinon ligne 1).

##### 5.1.1.2.2. Exécution des tours

On exécute tour à tour les créatures de la liste exécutive.
Une fois arrivé à la fin de la liste, on renvoie qu'un tour a été joué.

### 5.1.1.2.3. Génération d'une créature

On tire au sort une créature déjà créée parmi la population

On copie son code dans la nouvelle espèce.

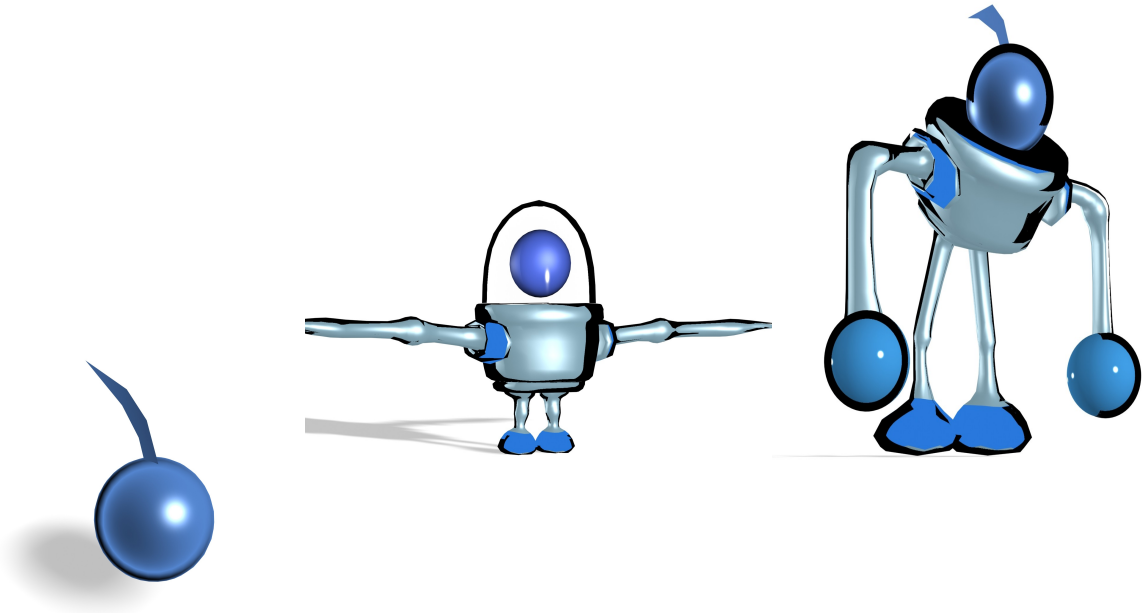
Tant que le nombre L de ligne modifiées n'a pas été atteint :

On tire une ligne au sort par mis les lignes existantes.

Si cette ligne est modifiable, on tire au sort une instruction par laquelle on doit la remplacer.

Sinon, on retourne au début de cette boucle.

Une fois les lignes modifiées, on retourne l'espèce (son pointeur).



### 5.1.1.2.4. Calcul du fitness

Pendant la partie, lorsqu'une créature appartient à notre camp et qu'elle en contamine une autre, alors elle accumule des points. Lorsqu'elle meurt, elle en perd.

A la fin de la partie, nous convertissons ces points en une note appelée fitness, un entier calculé sur l'accumulation de l'ensemble des créatures qui ont joué pour nous.

### 5.1.1.2.5. Création de la liste exécutive

D'après la règle du jeu, nous avons *n* créatures de même espèce (donc même code) par équipe.

On retient donc en mémoire le pointeur qui pointe vers l'algorithme de la créature à faire évoluer. Et celui de la créature adversaire.

Tant qu'on n'a pas atteint le nombre *n*2* de cellules dans la liste :

On crée une nouvelle cellule avec le pointeur vers l'algorithme de la créature à faire évoluer, le nom de l'espèce à laquelle elle appartient et sa position.

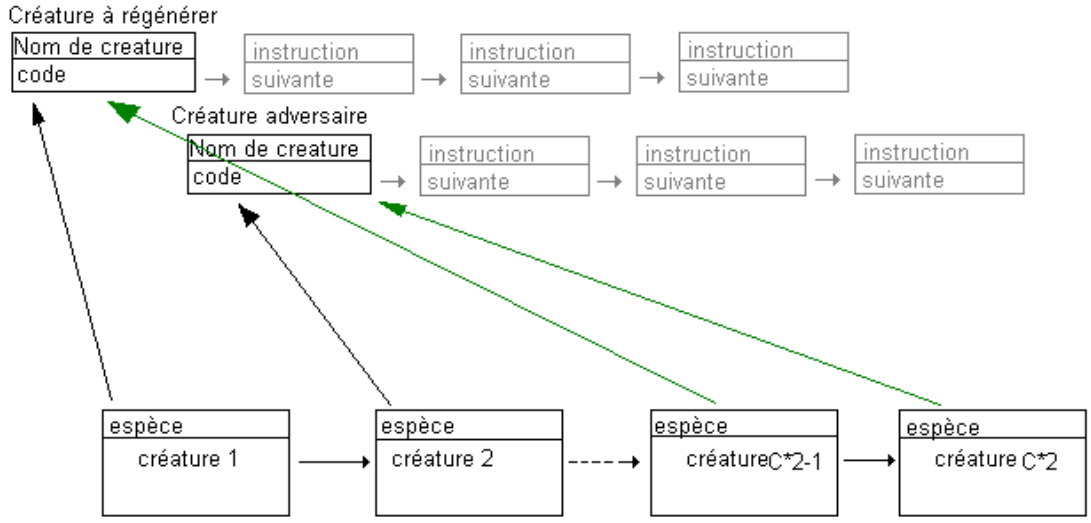
On crée une nouvelle cellule avec le pointeur vers l'algorithme de la créature adversaire, le nom de l'espèce à laquelle elle appartient et sa position.

Tant qu'on n'a pas inversé *n*2* cellules de la liste :

On tire au sort deux cellules et on inverse leurs pointeurs.

On renvoie la liste.

*Liste non mélangée :*



#### 5.1.1.2.6. Exécution d'une partie

On crée la liste exécutive.

Tant que le nombre de tours prédéfini n'a pas été atteint et qu'il reste des créatures dans chaque équipe, on exécute un tour.

A la fin d'une partie, on détermine le fitness de l'espèce qui a été générée.

On enregistre le code de l'espèce et son fitness.

#### 5.1.1.2.7. Tirage au sort d'une créature

On a une variable T qui nous donne le total de la somme des fitness des créatures. On tire au sort une valeur S de fitness réelle entre ]0;T[.

On parcours la liste des créatures en mémoire tant que la somme des fitness des créatures parcourues n'est pas égale à S.

La créature sur laquelle on va s'arrêter sera celle sélectionnée pour évoluer.

#### 5.1.1.2.8. Simulateur

Tant qu'il existe des fichiers adversaires :

On ouvre le fichier adversaire.

Tant qu'il existe des fichiers créature à régénérer :

On ouvre le fichier créature à générer.

Tant que le nombre de parties (et donc de mutations) n'est pas atteint :

On exécute la partie.

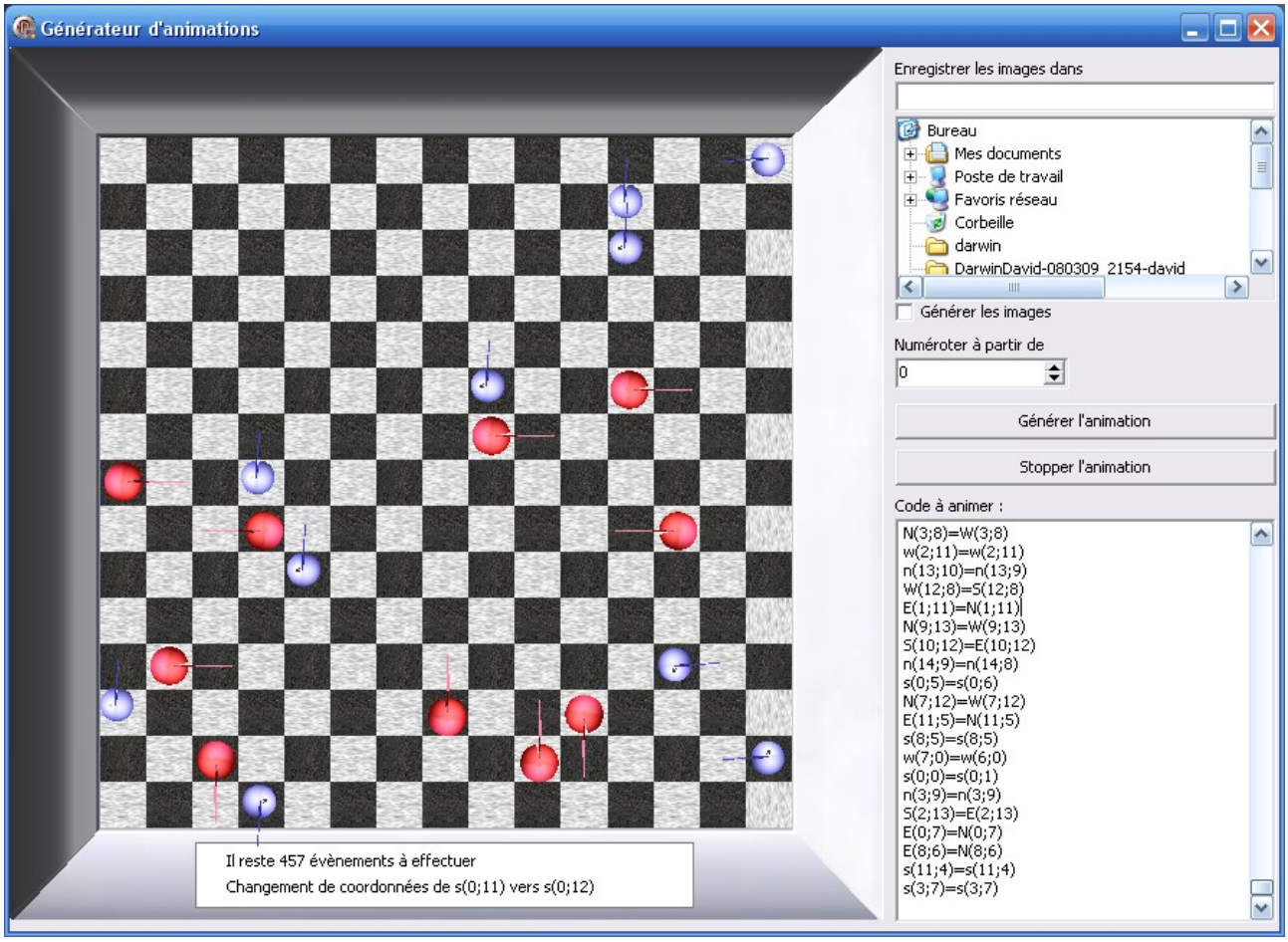
On enregistre l'espèce à garder.

On tire au sort l'espèce à rmuter dans le camembert

On la mute.

### 5.1.2. Programme d'animations

De façon à observer le déroulement d'une partie, nous avons fait en sorte que notre programme de simulation génère un script récupérable par un logiciel d'animation de notre conception. Nous pouvons alors visionner autant de fois qu'on le souhaite, les parties qui se sont jouées pendant la simulation.



### *5.2. Étude*

Les résultats obtenus ont démontré qu'au cours des multitudes d'évolutions, les créatures ayant obtenu les meilleurs fitness sont doté d'une stratégie assez surprenante lorsqu'on pense qu'elles ont été obtenues par le hasard. On constate par exemple qu'elle font en sorte de faire toutes les vérifications possibles avant d'agir.

D'autres créatures parcourent le damier dans un sens défini puis infectent les créatures ennemies. Elles peuvent donc infecter plus facilement  les créatures qui se déplacent peu.

Une autre génération de créatures que nous avons observé infectent en priorité l'ennemi, mais se déplacent rarement sur le damier. Ainsi elle ont plus de chance de pouvoir infecter une créature ennemie qui irait dans leur sens.

Un problème que nous avons relevé en revanche est que parfois, le hasard fait mal les choses, et au bout de quelques heures de simulations, aucune stratégie n'émane de la créature obtenue.

Dans l'optique de comparer plusieurs méthodes d'évolution, nous avons implémenté deux programmes, l'un respectant les règles évolutionnistes, et l'autre qui générait bêtement des espèces pour les noter.

## 5.2.1. Programme non évolutionniste

#### *5.2.1.1. Fonctionnement*

Cet algorithme part d'une créature déjà écrite. Il évalue la première espèce, la mute, l'évalue, sélectionne aléatoirement l'une des espèces créées pour la muter à son tour et ainsi de suite.

#### *5.2.1.2. Résultats*

Nous nous sommes rendu compte que la convergence de cet algorithme fonctionnait, mais qu'on retrouvait le programme de départ légèrement modifié classé en tête des fitness.

Exemple de code de créatures :

```
Flytrap_48881       Flytrap_183101      Flytrap_24103
ifenemy 4           left 8              infect 2
infect 8            ifempty 5           left
left 6              ifsame 5            ifenemy 5
infect              infect              ifenemy 5
go 1                hop 4               go 1
ifenemy 2           ifenemy 4
ifempty 3           left
go 1                go 1
hop 10              left 4
right 3             ifenemy 4
```

## 5.2.2. Programme évolutionniste

### *5.2.2.1. Fonctionnement*

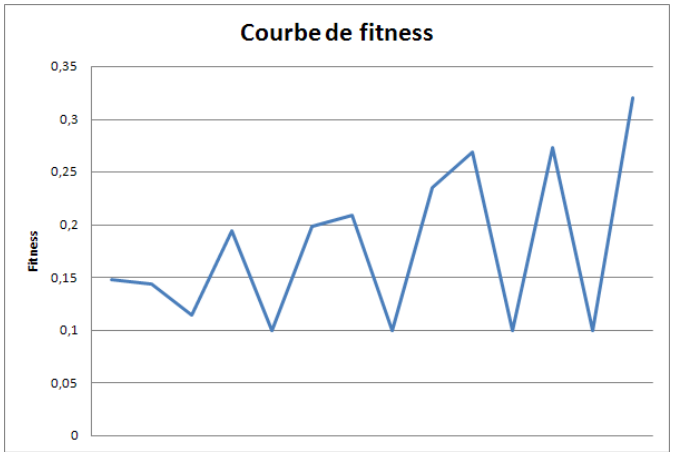
Comme l'autre programme, il est divisé en deux parties :

- Le jeu : Appelé par la fonction « Play » qui évalue une espèce donnée et renvoie son fitness.
- L'évolution : Elle se charge de générer la première population, de récupérer son fitness, et de générer la suivante à partir de la première.

  La suivante est générée à partir des espèces des générations précédentes. Pour savoir quelle espèce il va muter, le programme la sélectionne aléatoirement en favorisant les espèces aux plus forts fitness.

Cette méthode évolutionniste a un gros défaut, elle stocke en mémoire toutes les espèces générées, ce qui rend la sélection de moins en moins efficace. Il est alors de plus en plus difficile pour les espèces, de converger vers le fitness maximum.
Il nous est même souvent arrivé que le programme soit stoppé directement par le système d'exploitation en raison de la demande trop importante de ressources.

### *5.2.2.2. Résultats*




Nous avons lancé plusieurs dizaines de simulations sur notre programme et sommes arrivés à des résultats très intéressants.

Voici l'exemple de Gen9053_espece_71

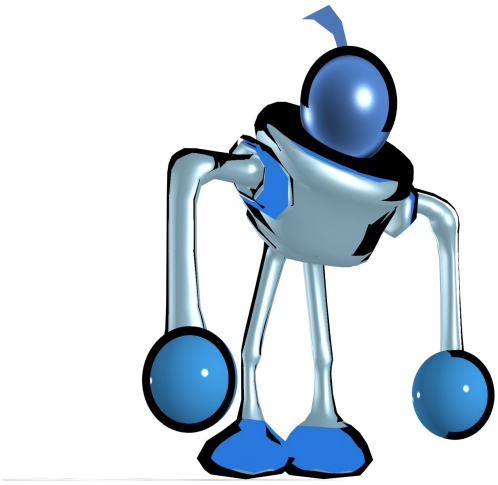
1 : ifenemy 9
2 : ifsame 8
3 : left
4 : ifrandom 2
5 : ifwall 6
6 : infect
7 : infect
8 : ifwall 10
9 : ifenemy 6
10 : hop

En l'analysant, nous nous rendons compte de plusieurs choses ; la première est qu'il a compris que s'il rencontrait un ennemi, il devait exécuter la même action ; la première ligne « ifenemy » renvoie donc à la neuvième ligne « ifenemy » qui renvoie à la ligne 6 qui infect. Il a donc également compris qu'en infectant un ennemi, il gagnait des points.

Son objectif était de vaincre le programme PPD2008 suivant :

PPD2008

1 : ifenemy 12
2 : ifsame 8
3 : ifwall 6
4 : ifempty 10
5 : go 8
6 : right
7 : go 1
8 : left
9 : go 1
10 : hop
11 : go 1
12 : infect
13 : go 1



# Conclusion

Il n'existe pas de meilleur moyen de comprendre la complexité des algorithmes évolutionnistes qu'en en produisant un soi-même. La difficulté résulte dans la manière de gérer la très grande quantité d'informations produite par l'algorithme et le temps que prend une seule simulation pour parfois arriver à un résultat médiocre.

C'est pourquoi il a été intéressant d'exploiter trois façons de voir l'algorithme évolutionniste.

La première méthode étant de simplement faire évoluer une espèce prédéfinie un certain nombre de fois et de récupérer la plus apte à combattre.

La deuxième est de générer une population initiale de façon totalement aléatoire pour en tirer le plus fort à chaque mutation. Cette étape est réitérée un nombre de fois fixé au départ en reprenant l'espèce alpha comme base de population.

La différence entre la troisième méthode et la seconde réside dans la façon de gérer la population. En effet, cette dernière méthode ne remplace pas la totalité de la population mais ajoute les espèces mutées à celle-ci. Nous n'avons malheureusement pas eu le temps de mettre en place cette méthode.

Reste à imaginer le les innombrables possibilités offertes par ce type d'algorithme dans la recherche de solutions à des problèmes insolvables jusqu'à aujourd'hui.

# Bibliographie

**Livres utilisés :**

*Les systèmes intelligents basés sur la connaissance,* 1988, par W.J.Black, Ed. Masson.

*Introduction à la conception des systèmes intelligents,* 1985, Igor Aleksander, Ed. Hermes

*L'origine des espèces,* Charles Darwin

**Internet :**

A Frog Norn from : <http://www.gamewaredevelopment.co.uk/creatures_more.php?id=465_0_6_0_M8>, trouvé le 26/10/2007

Portrait de Charles Darwin : <http://fr.wikipedia.org/wiki/Charles_Darwin>

Schéma fonctionnel : <http://fr.wikipedia.org/wiki/Algorithme_g%C3%A9n%C3%A9tique>

**Historique :**

<http://fr.wikipedia.org/wiki/Algorithme_%C3%A9volutionniste>

<http://www.etis.ensea.fr/~revel/html/cours_IA/node4.html>
<http://artis.imag.fr/Members/Gilles.Debunne/Enseignement/RICM/PDF/IA.pdf>
<http://sis.univ-tln.fr/~tollari/TER/AlgoGen1/node5.html#SECTION00052000000000000000>

**Damier :**

<http://nifty.stanford.edu/darwin/darwin99.pdf>

# Table des matières

# Annexes

# CD-Rom

Voir le contenu du CD-Rom fourni avec le rapport PPD avec :

Le programme non évolutionnistes

Le programme évolutionnistes

Le programme d'animation

L'affiche

Les images utilisées

La vidéo

Résumé de l'avancement du PPD

# Utilisation des méthodes évolutionnistes dans la découverte de stratégies


● Guillaume CHARTON (Classe 21) <charton@et.esiea.fr>
● Michaël DARMES (Classe 22) <darmes@et.esiea.fr>
● David LEBLANC (Classe 21) <leblanc@et.esiea.fr>
● Laurent MORILLON (Classe 21) <morillon@et.esiea.fr>


« Darwin's World » est un jeu de simulation d'un monde « vivant » qui met en compétition deux espèces de créatures. Le but de chaque espèce est de rallier les créatures ennemies à son camp. Nous avons repris les règles de ce jeu en considérant l'une des espèces comme l'ennemie dont le comportement ne change pas, et l'autre étant l'espèce à améliorer.

L'objectif du projet est d'implémenter en C, un algorithme évolutionniste qui fait évoluer l'espèce pour battre l'ennemi à partir du résultat obtenu par les créatures générées au cours des parties.

A la date d'aujourd'hui, nous comptons environs 50 heures de travail durant lesquelles nous avons :
- définit une méthode évolutionniste
- construit un schéma fonctionnel du programme
- compléter du cahier des charges
- écrit un programme capable de simuler le « monde de Darwin »
- écrit qui récupère des données du précédent programme afin de générer une animation

L'implémentation du principe évolutionniste est en cours et sera bientôt terminée. Lorsque notre programme sera complètement terminé, nous pourrons expliciter l'évolution de nos créatures à l'aide de graphiques.
Jusqu'ici le travail a plutôt bien avancé et nous sommes motivé pour arriver à un résultat fiable, cohérent et présentable.