ÉCOLE POLYTECHNIQUE
Promotion X-96
Rapport de stage d'option scientifique

Vito Trianni - Marco Salvi

Robotique Évolutionnaire : Apprentissage d'un Contrôleur Neuronal pour le Khepera

Option :	**Informatique** **Département de mathématiques appliquées**
Directeur d'option :	**Jean-Marc Steyaert**
Directeur de stage :	**Marc Schoenauer**
Lieu du stage :	**École Polytechnique** **CMAP**
Dates du stage :	**14 avril 1999 - 2 juillet 1999**

Résumé

Les Algorithmes Évolutionnaires ont reçu beaucoup d'attention grâce a leur potentialité comme techniques d'optimisation pour problèmes complexes. Le but que l'on s'est posé est d'apprendre la meilleure topologie et les meilleures paramètres pour un réseau de neurones afin de l'employer comme contrôleur pour un robot mobile, le Khepera. L'apprentissage a été fait par les Algorithmes Évolutionnaires, en partant de le problème plus simple d'approcher avec un réseau de neurones une fonction a deux variables, pour arriver à donner un contrôleur neuronal au robot.

Evolutionary Algorithms have received a lot of attention regarding their potential as optimization techniques for complex problems. Our aim is learning the best topology as well as the best weight setting of a Neural Network for controlling a mobile robot, the Khepera. The learning method is performed by Evolutionary Algorithms : we start resolving an easy problem of two variable function approximation and we finish describing the learning method of the neural controller for the robot.

Table des matières

Introduction

Dans les dernières années, plusieurs chercheurs ont essayé d'éviter de donner une structure fixée aux architectures des contrôleurs pour des robots mobiles qui doivent avoir un certain comportement dans n'importe quel environnement ; problème difficile, car on ne peut ni prédire tous les problèmes que le robot doit résoudre, ni tous les environnements possibles dans lesquels il se trouvera ; de plus il manque en général de principes de base qui nous aident à réaliser un contrôleur pour un problème donné. Une nouvelle approche donc semble être prometteuse : *la robotique évolutionnaire*. En suivant cette approche, le contrôleur du robot est adapté à l'environnement et au problème spécifique que l'on veut résoudre, en utilisant un processus de sélection qui élimine les individus faibles et par contre favorise la génération de nouveaux individus par reproduction stochastique des meilleurs.

Ce processus peut être implémenté par différentes procédures, comme les algorithmes génétiques, les stratégies d'évolution, la programmation évolutionnaire ; il s'agit de faire évoluer une population de *génotypes*, l'information modifiée de génération en génération. Chaque génotype code un *phenotype*, c'est-à-dire ici l'architecture du contrôleur du robot. Une fonction de performance est utilisée pour évaluer le comportement de chaque individu dans la population et pour diriger la sélection des meilleurs. Des opérateurs tels que la mutation ou le croisement sont construits pour générer des nouveaux individus, de manière stochastique, en espérant avoir des performances toujours croissant, jusqu'à ce que le processus atteigne l'optimum, local ou global.

Dans ce travail nous avons essayé de trouver le meilleur contrôleur pour un robot, le Khepera, en employant des réseaux de neurones *feed-foward*. Les réseaux de neurones se sont montrés des outils très puissants pour de nombreux problèmes, (approximation de fonctions, reconnaissance de formes...) ; l'avantage principal est la possibilité de l'entraîner : on peut utiliser l'algorithme de rétro-propagation lorsqu'on dispose d'exemple, ou les algorithme génétique ou les stratégie d'évolution ; on a choisi la dernière méthode pour trouver la meilleure topologie du réseau afin que le comportement du robot s'approche de ce que l'on souhaite.

Ce mémoire est organisé de la façon suivante. Le chapitre 1 décrit les Algorithmes Évolutionnaires, le chapitre 2 les réseau de neurones, le 3 le matériel et les logiciels utilisé ; le 4 parle de l'apprentissage d'un réseau pour approcher une fonction mathématique, le 5 de l'apprentissage d'un contrôleur pour le Khepera et le 6 conclue le mémoire.

Chapitre 1

L'évolution Artificielle

Depuis une trentaine d'années l'idée de réaliser une optimisation inspirée au processus naturel de l'évolution a amené plusieurs chercheurs à en développer une théorie simplifiée, les *algorithmes évolutionnaires* [1]. Ils sont basés sur le concept de population, où chaque individu représente un point dans l'espace de recherche ; la population évolue de génération en génération jusqu'à atteindre l'optimum ; des opérateurs de *mutation*, *récombination*, *sélection* nous permettent de générer une nouvelle population en partant de la population courante, un essai de simuler le processus de développement et d'adaptation d'une population réelle via reproduction, croisement et mutation de gênes et sélection naturelle des meilleures individus.

Faire une optimisation avec des algorithmes évolutionnaires se justifie lorsqu'on n'a pas de méthodes classiques d'optimisation qui marchent bien. Toutefois quand le problème est très complexe, on peut choisir soit de le simplifier pour obtenir une fonction objectif dérivable, et utiliser les méthodes classiques, soit de résoudre directement le problème initial avec les techniques évolutionnaires qui sont d'ordre 0 (n'ont besoin que de calculer la fonction à optimiser). Le problème principal est la complexité de l'algorithme, qui d'habitude a besoin de beaucoup d'évaluation de la fonction objectif ; de plus on n'est pas sûr d'avoir la convergence de l'algorithme dans tous les cas. L'avantage principal est d'avoir un modèle standard qui peut être adapté à la majorité des problèmes d'optimisation.

1.1 Le Schéma d'Évolution

Nous présentons ici une brève description du concept général d'un algorithme d'optimisation basé sur les techniques évolutionnaires ; pour une approfondissement de la matière, voyez [1, 2, 3]. Soit E l'espace de recherche, F la fonction a optimiser telle que $F : E \to \Re$; on considère le problème d'optimisation comme une maximisation sans perdre en généralité ; on travaille sur une population d'*individus*, c'est-à-dire de point dans E ; pour chaque individu on calcule sa *fitness*, une valeur qui donne une appréciation de la performance de l'individu et qui en général est égale à la fonction F ; en employant les opérateurs de mutation et croisement la population génère des autres individus, et elle est remplacée par l'opérateur de sélection. Un algorithme de base peut être le suivant :

```
Algorithme AE

t ← 0
initialisation de la population
évaluation de la population
while( ! condition-de-fin )
{
   t ← t + 1
   seléction des individus pour la reproduction
   opérateurs(mutation, récombination)
   évaluation des nouveaux individus
   remplacement de quelque parents avec quelques descendants
}
```

L'initialisation est faite d'habitude aléatoirement, mais il est possible de commencer par une population que l'on déjà appris quelque chose ; l'évaluation est faite par la fonction F ou par une fonction de performance différente, basée sur le calcul de F ; en fait, on peut remarquer une différence entre ce que l'on appelle “phénotype” et le “génotype” : le phénotype est un point de l'espace de recherche E, le génotype est la représentation codée du phenotype de façon telle que l'on puisse employer des algorithmes de reproduction spécifiques ; c'est pour ca que souvent F et la fonction de performance ne sont pas toujours identiques.

Ce qui détermine l'algorithme évolutionnaire est la façon de faire la sélection, la mutation et la récombination, et aussi le codage du phénotype en génotype ;

les différentes approches ont amené à la spécialisation en branches historiques qui maintenant sont très stables : *Algorithmes Génétiques* (GAs), *Stratégie d'évolution* (ESs), *Programmation Évolutionnaire* (EP). Dans les sections suivantes on explique les caractéristiques de chaque technique [1].

1.1.1 Algorithmes Génétiques

Dans la version canonique, les individus sont de chaînes de bit ($E = \{0, 1\}^n$) donc on doit coder l'information pour s'adapter ; les nouvelles versions utilisent un espace de recherche quelconque, sans modifier le concept de base ; dans tout les cas la version originelle est assez générale puisqu'il est souvent possible coder le phénotype dans une chaîne de bit.

- **Sélection** : le schéma est basé sur une probabilité d'être sélectionné mesurée par la fitness de l'individu (la méthode la plus connue étant la *roulette wheel selection*). Ceci nous permet d'explorer l'espace de recherche le plus que possible, parce que la sélection n'est pas déterministe (les individus les plus mauvais ont toujours une probabilité non nulle d'être sélectionnés).
- **Mutation** : On change la valeur d'un bit de l'individu avec une probabilité p_n qui est d'habitude très petite (de l'ordre de 10^{-3}).
- **Récombination** : cet opérateur est normalement implémenté par le *croisement* : en général on part de μ parents pour générer un seul enfant ; si on considère $\mu = 2$ pour simplicité, on fait le croisement en prenant une partie du premier parent et une partie du deuxième ; une autre façon est de décider si choisir pour le premier parent ou pour le deuxième à chaque bit du nouveau individu. Il est facile d'étendre tout ça pour $\mu > 2$.

1.1.2 Stratégie d'Évolution

L'espace de recherche E peut être quelconque, originalement on avait un vecteur de réels, mais l'extension a été immédiate. Il gère une population de μ individus qui se reproduisent en générant λ enfants : en moyen tout parent génère λ/μ individus par les opérateurs.

- **Sélection** : La façon de faire la sélection est tout à fait déterministe, car on choisit toujours les meilleurs individus pour former la prochaine population. Le type de choix détermine les caractéristiques de l'algorithme : on peut choisir la nouvelle population en prenant μ individus parmi les λ enfants

((μ, λ)-ES) ou parmi l'ensemble des parents et des enfants (($\mu + \lambda$)-ES). Dans tous les deux cas λ doit être plus grand que μ (une heuristique souvent utilisée est $\lambda = 7\mu$). ($\mu + \lambda$)-ES est une stratégie *élitiste*, c'est-à-dire que le meilleur individu reste toujours dans la population actuelle, à la différence de (μ, λ)-ES.

– **Mutation** : Cet opérateur donne la caractéristique de 'auto-adaptation' de l'algorithme : en travaillant avec des vecteurs réels $R = r_1, .., r_n$, on ajoute un vecteur de déviations standard $S = \sigma_1, .., \sigma_n$ qui fait partie de l'individu ; on fait muter chaque valeur r_i en suivant une loi gaussienne, $N(0, \sigma_i)$, après avoir muté σ_i aussi [1] ; dans un cas plus général la mutation peut être un opérateur quelconque qui change l'individu pour explorer de nouvelles zones de l'espace de recherche.

– **Récombination** : la récombination peut être fait de plusieurs façons, en suivant une approche classique (deux parents qui génèrent un individu) ou une approche *globale*, c'est-à-dire que le nouveau individu est créé en prenant l'information de toute la population. Il reste quand même un opérateur moins important que la mutation, même si souvent on ne peut s'en passer.

On peut démontrer la convergence d'une version simplifiée des ESs, (1+1)-ES et après il est facile d'étendre ce résultat à (($\mu + \lambda$)-ES) ; par contre, la convergence de ((μ, λ)-ES) n'est pas démontraée, même si maintenant elle reste la méthode la plus utilisée entre les deux.

1.1.3 Programmation Évolutionnaire

Ce schéma n'utilise que de récombination, et la mutation et faite d'une façon similaire à celle des ESs : la déviation standard est obtenue de la racine carré d'une transformation linéaire de la fitness. Ce qui change est la **sélection**, qui maintenant n'est plus déterministe mais est faite par un tournoi stochastique entre parents et enfants : étant donné la taille du tournoi q, on choisit q individus et pour chacun on le compare avec les autres en regardant la fitness, en gagnant un point quand il a la performance la meilleure. À la fin de ce processus tous les individus ont un score entre 0 et q, et on prend les individus avec le score le plus haut. Cet algorithme est donc élitiste, puisqu'il est sûr que le meilleur aura q point après le tournoi. Si la taille du tournoi grandit, l'algorithme devient semblable à un ($\mu + \mu$)-ES, pour être tout à fait identique si $q = \mu + \lambda$.

1.2 EvolC

Cette section est un extrait du 'Manuel d'utilisation du programme EvolC (Marc Schoenauer - Février 1998)'.

EvolC est un environnement de développement d'algorithmes évolutionnaires réalisé au CMAP. Écrit en C++, les points les plus importants en sont :

- De très nombreux paramètres facilement modifiables (ce n'est un point fort que parce qu'il n'existe pas aujourd'hui de valeurs universelles pour tous ces paramètres, hélas).
- Possibilité de faire évoluer des types de représentation classiques (chaînes de bits et vecteurs de réels) avec leurs opérateurs usuels en n'écrivant que la fonction fitness, mais également de rajouter des représentations quelconques en ne réécrivant que le minimum de routines spécifiques (initialisation aléatoire, recopie, opérateurs génétiques et entrées/sorties).

[...] Une lente évolution a alors abouti au programme EvolC, dans lequel, grâce au mécanisme d'héritage du C++, a pu être programmé un algorithme à peu près général (quoiqu'il reste encore du chemin à parcourir !), manipulant deux types d'abstraction : un schéma d'évolution, regroupant l'ensemble des processus de sélection au cours d'une génération, et un problème qui regroupe ce qui est spécifique au problème traité (représentation et opérateurs associés et fonction fitness). Le pseudo-code suivant donne une idée de l'algorithme, signalant entre crochets les parties spécifiques.

```
choose scheme and first "template" individual}
randomly initialize the population            <problem-specific>
evaluate each individual                      <problem-specific>
do forever
   statistics
   display                                    <problem-specific>
   stopping criterion ?
      if yes then end
   until enough offspring do
      initial selection of the first parent   <scheme-specific>
      choose evolution operators              <problem-specific>
      if crossover,
```

```
      select mate(first parent)                <scheme-specific>
    apply evolution operator                   <problem-specific>
    evaluate each newborn offspring            <problem-specific>
    select between parent(s) and offspring(s)  <scheme-specific>
  enddo
  replace the parents                          <scheme-specific>
end
```

La partie <`scheme-specific`> peut être décrite en détail en utilisant les paramètres disponibles.

[...] Un ensemble de paramètres décrit les divers schémas d'évolution disponibles au sein de EvolC. Cela recouvre de nombreux schémas classiques (AG traditionnels, SSGA, (μ, λ)-ES, et $(\mu + \lambda)$-ES, EP) mais permet également de créer sa propre variante de schéma d'évolution à l'aide de sélecteurs divers et variés.

[...] Pour leur nature, les algorithmes évolutionnaires sont dépendants de tirages de nombres aléatoires. C'est pourquoi une règle d'or est de ne jamais tirer de conclusion d'un unique essai. Il faut donc lancer un certain nombre d'essais pour lesquels tous les paramètres sont identiques sauf la graine du générateur de nombre aléatoire . On examine ensuite des statistiques sur ces essais, et il est alors possible de tirer certaines conclusions.

Les statistiques en question sont le plus souvent des moyennes+écart-types (ou des médianes) de la meilleure performance au cours de l'évolution, accompagnées éventuellement de tests de pertinence statistiques. Il faut cependant être bien conscient que ces tests supposent implicitement que la répartition des résultats suit une loi normale, ce qui n'est pas forcément justifié.

Deux utilitaires sont fournis, *depouille*, qui génère des statistiques sur un ensemble de fichiers texte de même structure, et *ttest*, qui valide les comparaisons entre des sorties de depouille faites dans des conditions expérimentales différentes.

Chapitre 2

Réseaux de Neurones

2.1 Introduction

Les **réseaux de neurones formels** sont à l'origine une tentative de modélisation mathématique du cerveau humain. Les premiers travaux datent de 1943 et sont l'oeuvre de MM. Mac Culloch et Pitts. Ils présentent un modèle assez simple pour les neurones et explorent les possibilités de ce modèle. Un réseaux de neurones peut être vu comme un groupe de simples unités (neurones) dans lesquelles la structure et le type de la connections définissent la capacité de calcule du réseau.

2.2 Le modèle biologique

Dans le cerveau, les neurones sont reliés entre eux par l'intermédiaires d'axones et de dendrites. En première approche on peut considérer que ces sortes de filaments sont conducteurs d'électricité et peuvent ainsi véhiculer des messages depuis un neurone vers un autre. Les dendrites représentent les entrées du neurone et son axone sa sortie.
Un neurone émet un signal en fonction des signaux qui lui proviennent des autres neurones. On observe en fait au niveau d'un neurone, une intégration des signaux reçus au cours du temps, c'est-à-dire une sorte de sommations des signaux. En général, quand la somme dépasse un certain seuil, le neurone émet à son tour un signal électrique.

La notion de synapse explique la transmission des signaux entre un axone et une dendrite. Au niveau de la jonction (c'est-à-dire de la synapse), il existe

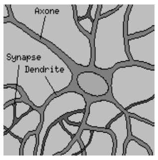


FIG. 2.1: Un neurone biologique

un espace vide à travers lequel le signal électrique ne peut pas se propager. La transmission se fait alors par l'intermédiaire de substances chimiques, les neuro-médiateurs. Quand un signal arrive au niveau de la synapse, il provoque l'émission de neuro-médiateurs qui vont se fixer sur des récepteurs de l'autre côté de l'espace inter-synaptique. Quand suffisamment de molécules se sont fixées, un signal électrique est émis de l'autre côté et on a donc une transmission. En fait, suivant le type de la synapse, l'activité d'un neurone peut renforcer ou diminuer l'activité de ces voisins. On parle ainsi de synapse excitatrice ou inhibitrice.

2.3 Le neurone formel

2.3.1 Le modèle

Le modèle de neurone formel présenté ici, dû à Mac Culloch et Pitts, est un modèle mathématique très simple dérivé d'une analyse (elle aussi assez simple) de la réalité biologique.

On constate tout d'abord que le modèle biologique fait intervenir une notion temporelle qui est difficile à intégrer dans un modèle simple. On oublie donc cette notion et de ce fait on remplace l'intégration temporelle par une simple sommation des signaux arrivant au neurone (ces signaux sont communément appelés les entrées du neurones). On compare ensuite la somme obtenue à un seuil et on déduit de la comparaison la sortie du neurone. Cette sortie sera par exemple égale à 1 si la somme est supérieure au seuil et à 0 dans le cas contraire. Plus formellement encore, il suffit pour obtenir ce comportement de soustraire le seuil considéré à la somme des entrées, et de faire passer le résultat par la fonction de transfert du neurone qui est ici la fonction de Heaviside. Le résultat après transfert est alors la sortie du neurone. Cette enchaînement "sommation" puis "non-linéarité" représente finalement les propriétés "physiques" du neurone. On peut améliorer la fonction de transfert en remplaçant la fonction de Heaviside par un *rampe* ou encore mieux par une *sigmoïde* modelé avec un paramètre β de la façon suivante : $\sigma(t) = \frac{1}{1+e^{-\beta t}}$ (fig. 2.2.

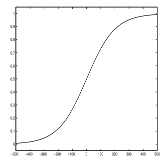
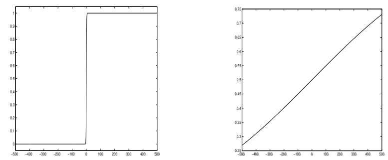


FIG. 2.2: Sigmoïde

Le paramètre $|\beta|$ représente la pente de la courbe au point $t = 0$; on peut remarquer que si $|\beta|$ est petit on aura une fonction très proche à la rampe (fig. 2.3(a)) , par contre, si $|\beta|$ est grand ($|\beta| \to \infty$) on aura presque une fonction de Heaviside (fig. 2.3(b)).



(a) heaviside (b) rampe

FIG. 2.3: Le sigmoïde en fonction du paramètre *beta*

La structure obtenue en rajoutant le sigmoïde est souvent appelée *neurone à état continu.*

La modélisation des synapses est assez simpliste en première approche. On se contente en effet d'introduire la notion de synapse excitatrice et de synapse inhibitrice en multipliant la sortie d'un neurone par 1 ou -1 avant de la transmettre aux autres neurones. Afin de donner plus de puissance au modèle, on généralise ce modèle en introduisant ensuite une connexion synaptique qui est une valeur réelle. La sortie d'un neurone est alors multipliée par la valeur de la connexion synaptique avant d'être transmise à un autre neurone. On modélise ainsi la connexion entre deux neurones.

En résumé, un neurone formel réalise simplement une somme pondérée de ces entrées, ajoute un seuil à cette somme et fait passer le résultat par une fonction de transfert pour obtenir sa sortie.

2.3.2 L'utilisation

On considère un neurone formel qui a le Heaviside pour fonction de transfert. Pour fonctionner, le neurone formel utilise des entrées qui sont des grandeurs réelles. Si on "relie" un neurone au monde extérieur par l'intermédiaire de capteurs, il peut réaliser une simple analyse de ce qu'il "perçoit". Si on représente les valeurs observées par le neurone sous forme d'un vecteur, le neurone réalise alors un découpage de son espace d'entrée (l'espace vectoriel auquel appartient le vecteur d'observation) en deux zones : la zone d'activité dont les vecteurs donnent une sortie égale à 1 et la zone d'inactivité dont les vecteurs donnent une sortie égale à 0. Comme le calcul effectué est en fait linéaire, la séparation l'est aussi. Les coefficients synaptiques et le seuil définissent l'équation d'un hyperplan qui est la frontière de la séparation entre les deux zones. En fait, un unique neurone est particulièrement limité car il ne sait calculer qu'une séparation linéaire.

2.4 Un réseau de neurones

Utiliser un neurone pour faire une séparation linéaire est assez décevant. Les possibilités d'un tel objet sont extrêmement limitées. On se demande alors si on ne peut pas imiter plus fidèlement la structure du cerveau en faisant travailler ensemble plusieurs neurones. On arrive à ça en connectant entre eux plusieurs neurones.

2.4.1 Typologies des réseaux

On peut commencer la description des typologies des réseaux de neurones par les réseaux multi-couche(RMC) (fig. 2.4). L'idée principale est de grouper des neurones par couche. En place ensuite bout à bout plusieurs couches et connecte complètement les neurones de deux couches adjacentes. Les entrées des neurones de la deuxième couche sont donc en fait les sorties des neurones de la première couche. Les neurones de la première couche sont reliés au monde extérieur et reçoivent tous le même vecteur d'entrée (c'est en fait l'entrée du réseau). Ils calculent alors leurs sorties qui sont transmises aux neurones de la deuxième couche, etc. Les sorties des neurones de la dernière couche forment la sortie du réseau.
Une généralisation des RMC sont les réseaux **feed forward** (RFF) qui sont

caractérisé par la particularité que le signal d'activation des neurones marche dans un seule direction. Généralement un RFF est caractérisé par un niveau d'entrée, un niveau caché et un niveau de sortie. Dans le niveau caché on peut avoir des connections, mais il ne faut pas avoir des boucles. De façon plus rigoureuse on peut imaginer de faire un ordonnancement partiel sur les neurones en le divisant en trois groupes :

– Les neurones d'entrées qui peuvent avoir des connections vers ceux cachés et vers ceux de sortie, mais non entre eux ;
– Les neurones cachés qui forment un ensemble ordonné et ils peuvent se connecter aux neurones des sortie et aux neurones cachés si on respecte l'ordonnancement ;
– Les neurones de sortie qui ne peuvent avoir des connections que vers l'extérieur.

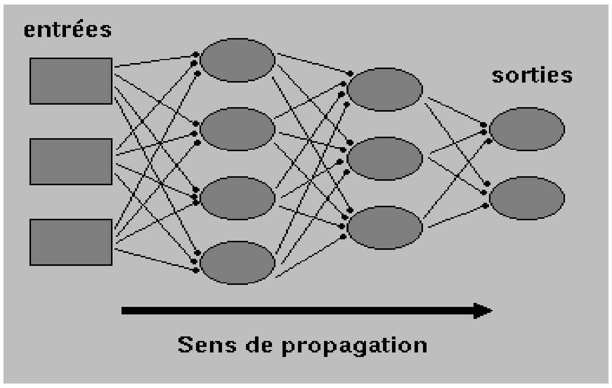


Fig. 2.4: Réseau multi-couche

La dernière typologie des réseaux que l'on va décrire est celle des Réseaux **récurrent** qui ont la particularité d'avoir des connections récursives. Ca veut dire que les entrées des neurones peuvent dépendre d'une sortie qu'ils ont générée eux-mêmes, c'est-à-dire qu'il y a des boucle dans le graphe qui représente le réseau.

2.4.2 L'apprentissage supervisé

Les considérations que l'on fera dans cette section et dans la prochaine, concernent en particulier les RMC et les RFF, mais on peut l'étendre, en apportant des variations, aux autres typologies des réseaux. En ajustant les paramètres d'un RN, on peut lui faire calculer toute sorte de fonction. Étant donnée une fonction vectorielle particulière, on peut tenter de faire apprendre cette fonction par un RN : c'est l'apprentissage. La méthode classique pour l'apprentissage supervisé consiste à se donner un ensemble d'*exemples*, c'est-à-dire un ensemble fini de couple de vecteurs (x_i, y_i). Dans un tel couple, x_i désigne l'entrée du réseau et y_i la sortie désirée pour cette entrée. On écrit alors la fonction calculée par le réseau sous une forme paramétrique : $f(x, w)$ désigne la sortie du réseau quand on lui présente en entrée le vecteur x et qu'il utilise les poids synaptiques contenus dans le vecteur w. On se donne enfin une distance sur l'espace vectoriel de sortie, c'est-à-dire un moyen de mesurer l'erreur commise en un point par le réseau. Si cette distance est notée d, on cherche alors à trouver la valeur de w qui minimise la somme de l'erreur totale commise par le réseau, c'est-à-dire la somme des distances entre les sorties obtenues et les sorties désirées, c'est-à-dire somme des $d(f(x_i, w), y_i)$. Cette erreur est une fonction de w et on peut donc utiliser les techniques classiques d'optimisation de fonction pour trouver son minimum.

2.4.3 La rétro-propagation du gradient

Les algorithmes d'optimisation de fonction efficaces utilisent en général la différentielle de la fonction considérée (c'est-à-dire son gradient car elle est à valeurs réelles). Quand les fonctions de transfert utilisées dans les neurones sont différentiables, et quand la fonction distance est aussi différentiable, l'erreur commise par un RN est une fonction différentiable des coefficients synaptiques du réseau. L'algorithme de rétro-propagation permet justement de calculer le gradient de cette erreur de façon efficace : le nombre d'opérations (multiplications et additions) à faire est en effet proportionnel au nombre de connexions du réseau, comme dans le cas du calcul de la sortie de celui-ci. Cet algorithme rend ainsi possible l'apprentissage d'un RN quand on dispose d'exemples.
Il faut dire que la rétro-propagation c'est un algorithme lent et susceptible de se trouver qu'un optimum local.

Chapitre 3

Le robot Khepera

3.1 Le robot réel

Le Khepera est un mini-robot mobile circulaire dont le diamètre est de 55mm, l'hauteur de 30mm, et le poids de 70g ; il est équipé de deux roues et de deux balles en Téflon. Dans sa configuration base, il a huit capteurs infrarouge de proximité - six devant et deux derrière - qui peuvent aussi capter l'intensité de lumière. Les roues sont contrôlées par deux moteurs avec des encodeurs par accroissement qui bouge dans les deux directions.

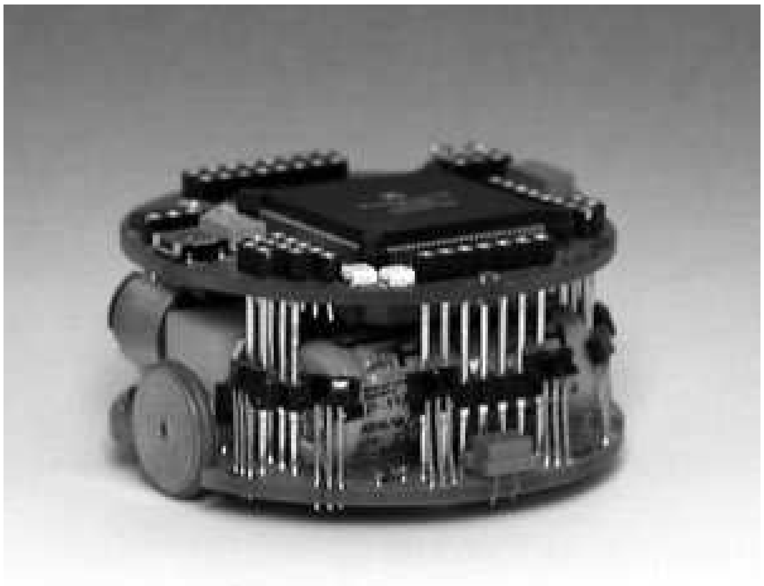


Fig. 3.1: le robot Khepera

Il a un processeur Motorola 68000 et il peut aussi être contrôlé par un ordinateur via une connection série. Chaque capteur de proximité a un émetteur et un récepteur : cela permet de mesurer deux choses : la lumière ambiante normale - par le seul récepteur - et la lumière réfléchie par les obstacles - en utilisant à la fois émetteur et capteur. Cela permet de détecter un obstacle situé a moins de 5cm si la lumière est moyenne ; par contre, si la lumière est trop forte, le capteur arrive a la saturation et on ne peut pas distinguer la lumière ambiante de la lumière de l'émetteur.

Le Khepera est équipé avec une RAM de 256 kilo-octet et un EPROM de 128 ou 256 kilo-octet : il est donc possible pour l'utilisateur de charger un contrôleur dans le robot ; en plus il est possible d'ajouter des modules différents comme une caméra, un 'gripper', un module pour la communication radio : tout ça rend le robot très versatile pour les différents possibilités de recherche. En fait, la petite dimension, la facilité de programmation et la possibilité d'ajouter plusieurs modules ont fait du Khepera l'instrument le plus utilisé pour expérimentation de robotique, y compris pour la robotique évolutionnaire.

3.2 Le simulateur

Le simulateur du robot est très important ; d'habitude le processus d'évolution est fait en deux phases : d'abord on cherche l'individu le meilleur avec une simulation spécifique, puis on le charge dans le robot réel pour avoir une idée de la performance de l'individu vue dans un environnement réel avec des contraintes bien sur différentes [6]. Toutefois il y a aussi des approches plus extrêmes : dans ses expériences, Mondada [4] fait exécuter le processus évolutionniste directement sur le robot réel : de cette façon il fait l'apprentissage d'un simple contrôleur neuronal pour éviter les obstacles. Par contre Jakobi [5] emploie les simulations á tout prix : il a développé un contrôleur neuronal pour un robot mobile à huit jambes en ne faisant que des simulations très simplifiées. Le problème est le suivant : on cherche un *contrôleur en boucle fermé* qui donne le déplacement du robot en fonction des informations de l'extérieur (capteurs, caméra, etc.). Par example, dans le cas le plus simple, on cherche un fonction de 8 entrées et 2 sorties qui donne le comportement voulu du robot lorsqu'elle le contrôle (voir fig. 3.2) ; les sorties de cette fonction gèrent les moteurs qui font deplacer le robot (un pas de temps), dont on a des nouvelles valeurs d'entrée pour la fonction. Pour le Khepera on peut utiliser deux simulateurs : la première version, *le khepera-simulator 2.0*, et la version nouvelle, *Webots*.

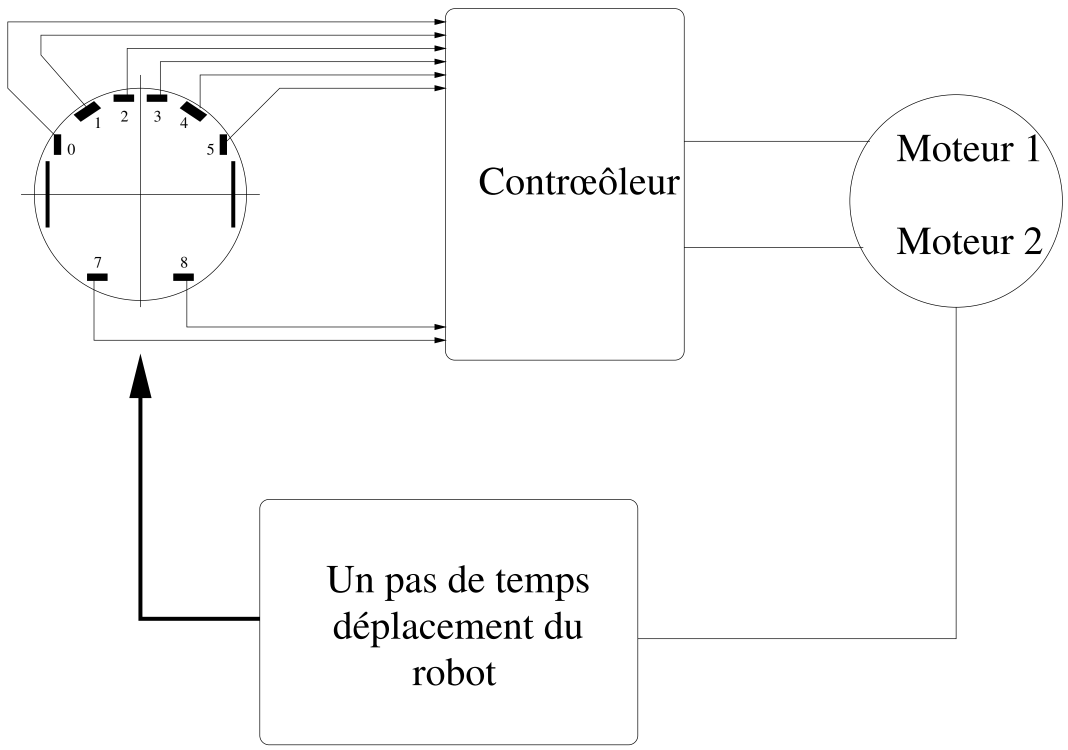


FIG. 3.2: Le contrôleur à boucle fermée.

Le Khepera simulator 2.0 fournit une interface graphique très simple qui nous permet de construire l'environnement dans lequel le robot devra bouger, de donner un contrôleur au robot et de visualiser n'importe quelle information dans des fenêtres.

Webots est la version commerciale du simulateur : c'est un simulateur de robot mobiles développé par Cyberbotics, pensé pour le chercheurs dans le domaine de l'intelligence artificielle, de la vision et des agents autonomes. Il est plus puissant que la première version : il simule le robot Khepera en 3D et tous les modules que l'on peut ajouter. Il arrive a modéliser assez bien les capteurs et les actuateurs, en donnant un comportement très semblable à la réalité, même si on peut remarquer de différences entre le comportement du robot simulé et du robot réel.

Le simulateur gère deux programme : le contrôleur et le superviseur, écrits par l'utilisateur. Le premier sert à donner le comportement souhaité au robot ; il a accès a toutes les fonction du Khepera, c'est-à-dire il peut lire tous le donnés qui caractérise l'état du robot (capteurs, vitesse des moteurs, position des roues) et donner une valeur de vitesse ou de position à chaque roue. De plus il peut contrôler aussi les modules supplémentaires éventuellement ajouté. Le superviseur est un programme qui permet de gérer des données concernant les robot et l'environnement en général (position du robot, position de la balle qui on peut ajouter pour simuler un match de football), en général de contrôler, sauvegarder ou afficher toute sorte d'information concernant l'environnement simulé. Une caractéristique très important est la possibilité de communication entre contrôleur du robot et superviseur ; l'échange d'information ayant des contraintes de dimension (à cause de la vitesse de communication), mais le type d'information que l'on peut envoyer est tout à fait libre : en théorie on peut même employer le superviseur comme un serveur de messages, afin que les différents robot qui sont dans l'environnement puissent communiquer entre eux pour atteindre un objective commun.

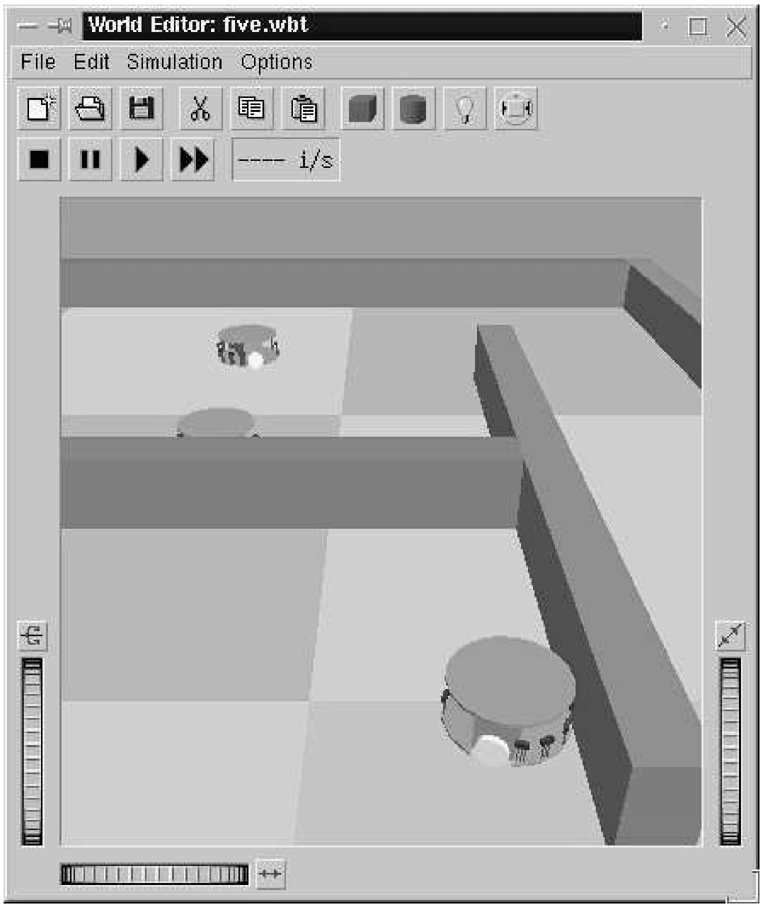


FIG. 3.3: Le simulateur Webots

Une autre caractéristique de *Webots* est le graphique, très bien fait en utilisant le VRML (*Virtual Reality Modeling Language*, mais elle donne des temps-calcul, limitant l'évolution artificielle du comportement du robot, lorsqu'on rajoute la caméra.

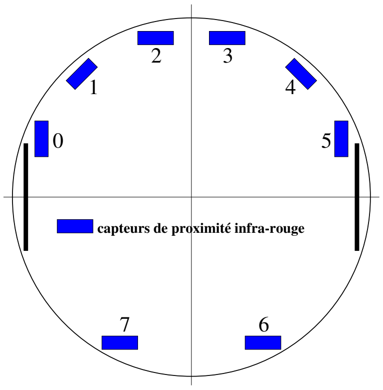


FIG. 3.4: Le schema du robot

.

Chapitre 4

Implantation de réseaux de neurones

4.1 Introduction

Dans ce chapitre on parle du travail que l'on a fait pour écrire des programmes pour la création et l'apprentissage de réseaux de neurones. On décrira les choix faits pour l'implémentation en parlant tout de suite de la méthode d'apprentissage que l'on a choisie pour nos projets et puis de l'apprentissage de réseaux multi-couche et de celui des réseaux feed forward.

4.2 La méthode d'apprentissage : les Algorithmes Évolutionnaire

Il y a plusieurs façons de faire l'apprentissage d'un réseau de neurones ; pour nos implémentations on a choisi de le faire par les Algorithmes Évolutionnaire. La rétro-propagation est un algorithme très précis et même assez rapide, mais le résultat dépend beaucoup de l'initialisation des poids et de la topologie choisie ; en fait il ne peut pas changer la topologie du réseau qu'il apprend, donc il faut de nombreux essais avec des initialisation différentes pour trouver la meilleure solution. De plus il ne peut pas marcher dans le cas où on n'a pas des exemples. La méthode Évolutionnaire par contre, même étant plus lente, n'a pas trop de contraints d'initialisation, peut optimiser la structure du réseau et n'a pas besoin d'exemples : elle s'avère le meilleur choix pour des problèmes complexes comme ces que l'on va traiter, c'est-à-dire la recherche d'un contrôleur pour un robot mobile.

On a envisagé le problème graduellement en commençant par approcher des fonctions à deux variables définies sur un compact.

4.3 Réseau multi-couche

Dans cette section on parle du travail que l'on a fait en étudiant l'apprentissage de réseaux de neurones multi-couche avec un nombre de neurones fixé.

4.3.1 L'implantation

On a implémenté le réseau en choisissant pour chaque neurone la structure de fig. 4.1 c'est-à-dire en le caractérisant par les paramètres suivants :

- Le *biais*, un paramètre que l'on rajoute toujours à l'entrée du neurone et on peut le considérer comme le poids de la connections d'un faux neurone dont la sortie vaut toujours 1 ;
- Le *beta*, qui modèle la fonction de transfert (voir 2.3) ;
- La fonction de transfert : un *sigmoïde* (voir 2.3).

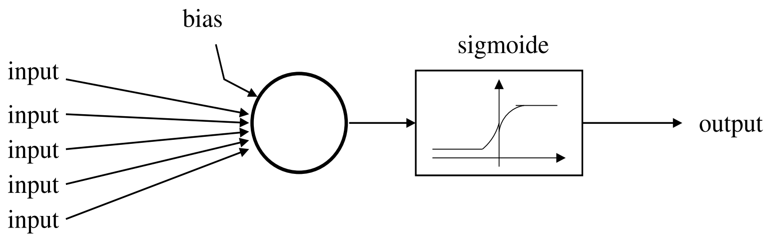


FIG. 4.1: Un neurone

Cette structure et tout à fait semblable au *neurone à état continu* que l'on a décrit dans la section 2.3.

L'ensemble des neurones est caractérisé par les paramètres suivants :

- Le nombre d'entrées
- Le nombre de niveaux cachés
- Le nombre de neurones des niveaux cachés

- Le nombre de sorties
- Le poids des connexions

Tous ces paramètres, à l'exception des poids des connexions, sont donnés au démarrage du programme et ne sont plus changés pendant l'évolution.

4.3.2 Apprentissage

L'apprentissage des réseaux a été réalisé par *EvolC* (voir 1.2), car il fournit des outils pour l'évolution d'un tableau de nombres réels. Puisque la structure du réseau est fixée au démarrage du programme on a pu codifier le phénotype "**réseau de neurones**" dans le génotype "**chaîne des réels**". Les chaîne que l'on fait évoluer contient donc les *biais* et les *beta* des neurones et les poids des connections synaptiques.

Le schéma d'évolution que l'on a choisi a été *ES* en faisant des essais avec *($\mu + \lambda$)-ES* et *(μ, λ)-ES* qui n'ont pas donné des résultats trop différant.

On a décidé de faire la mutation par auto-adaptation (voir 1.1.2) ; pour ce qui concerne le croisement, il prend deux vecteur de réels $\vec{a}(a_1, ..., a_n)$ et $\vec{b}(b_1, ..., b_n)$ et sort un vecteur $\vec{c}$ où les composantes sont a_i ou b_i $(0 \leq i \leq n)$ avec probabilité 0.5 (voir fig. 4.2).

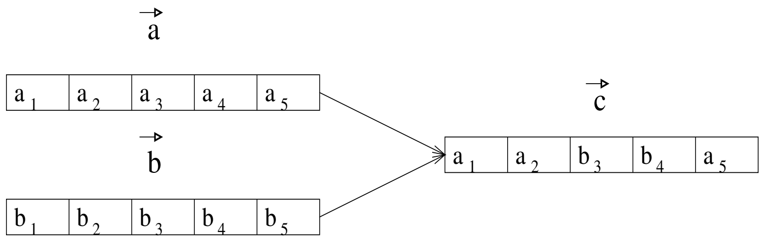


FIG. 4.2: Croisement

Comme on vient de dire l'apprentissage des réseaux se fait en leur faisant approcher une fonction mathématique ; par conséquence on a choisi comme fitness l'erreur carré moyenne entre la fonction donnée et celle calculée par le réseau, calculée sur une grille des 100 points.

4.3.3 Les tests

Les tests que l'on a effectués montrent que l'on peut approcher des fonction mathématiques avec des réseaux multi-couche. On est arrivé à obtenir une erreur carré moyenne inférieure à 10^{-3}.

On a faits des essais sur différents types de fonctions plus ou moins complexes par des différentes réseaux on s'est aperçus que les résultats sont fortement liés à la structure du réseau : par example une fonction simple n'a pas besoin d'un réseau avec beaucoup de neurones puisque ça peut compromettre le résultat. Il faut aussi remarquer qu'une structure fixée ne peut pas satisfaire la nécessité de résoudre des problèmes complexes pour lesquels on n'a pas d'exemples, comme celui de réaliser un contrôleur pour un robot. Pour ces raisons on a décidé de passer à une structure plus compliquée mais sûrement plus puissante qui nous permette d'avoir moins de contraintes vis-à-vis de la typologie du problème.

4.4 Réseaux Feed Foward à Topologie Variable

Dans cette section on décrit la façon d'apprendre la meilleure topologie et, bien sûr, les bonnes valeurs justes des paramètres qui caractérisent un réseaux de neurones. Comme on a déjà vu 2.4.1, un réseau feed foward est définitcomme un ensemble de neurones sur lequel on peut définir un ordonnancement : en suivant cette approche on a cherché la meilleure façon d'implémenter un réseau général.

4.4.1 Structure du Réseau

Il s'avère très difficile d'obtenir une structure qui soit simple à modifier et qui au même temps permette de faire le calcul des sorties. Ces deux objectifs sont assez contradictoires : le premier requiert un structure facilement modifiable, des pointeurs et des listes ; par contre le deuxième requiert une structure rapide à parcourir et donc fixée ; la meilleure solution est donc d'avoir deux structures différentes, chacune optimisée pour un but, et de faire en suite un passage de "compilation" en passant de l'une á l'autre.

Une structure optimisée pour le calcul doit être très simple : étant donné l'ordonnancement des neurones, on a déjà un ordre a suivre pour calculer les sorties, parce que les seules informations dont on a besoin pour calculer la sortie d'un neurone sont le sorties des neurones qui les précèdent dans l'ordonnancement. Il faut donc organiser les neurones dans un tableau en associant à chaque neurone les connexions vers lui même : le calcul de la sortie d'un neurone se fait en lisant les sorties des neurones qui lui sont connectés, en les multipliant par le poids synaptique de la connexion et en ajoutant le *biais* ; puis la somme pondérée ainsi obtenue est traitée a l'aide de la fonction de transfert.

Tout d'abord on peut remarqué qu'un réseau feed foward peut être formalisé comme un DAG (*Direct Acyclic Graph*), en rajoutant un faux neurone qui soit la racine de l'arbre et qui ait toutes ses sorties connectées aux neurones d'entrée. Cette structure est très générale et nous permet d'avoir n'importe quelle topologie et de réaliser n'importe quelle modification de celle-ci. Toutefois il est difficile d'en faire une traduction vers la structure optimisée pour le calcul qui soit performante : en fait la contrainte est d'avoir une complexité au plus de *O(n)*, puisque c'est une opération très fréquente. Avec le DAG on gagne en généralité mais on perd en vitesse. Comme on peut rattraper la généralité en employant les Algorithmes Évolutionnaires, on a choisi une structure plus simple qui ne nous permet pas de faire toute sorte de modifications mais qui nous donne une complexité de la traduction linéaire. Il s'agit d'une liste de neurones dont les connexions sont associées au neurone duquel elles partent : chaque connexion a l'information sur le poids et sur le neurone-destination, chaque neurone celle sur le *biais* et sur la *beta* (voir 2.3). Une connexion est légitime si le neurone-destination suit le neurone-départ ; de plus, on ne peut avoir de connexions ni entre les neurones d'entrée ni entre ceux de sortie.

4.4.2 Apprentissage

L'apprentissage se fait par des algorithmes évolutionnaires, mais cette fois on a un réseau entier à apprendre, pas seulement des paramètres ; on fait évoluer une population où chaque individu représente un réseau ; le nombre d'entrées et de sorties est constant, celui de neurones cachés varie entre un minimum et un maximum que l'on peut choisir. On initialise la population aléatoirement : il est important d'avoir une population très différenciée surtout au début quand on ne connaît pas la meilleure topologie ; l'opérateur d'initialisation agit choisissant de façon totalement aléatoire chaque paramètre du réseau : il choisit le nombre de neurons n du réseau au hasard, puis pour chaque neurones $n_i, 0 \leq i \leq n-1$ il choisit le nombre $m < n-i$ de connections à lui rajouter. Il essaie de les ajouter toutes au hasard, en s'arretant s'il ne le fait pas pour un nombre de fois fixé. Tous les paramètres sont initialisés avec des valeurs bornée, même si pendent l'évolutions il ne le sont pas.

On change la topologie du réseau pendant l'apprentissage par l'opérateur de mutation structurelle ; en plus on a réalisé un opérateur de croisement entre deux réseaux de neurones ; la technique évolutionnaire que l'on a choisi est la $(\mu + \lambda)$-ES : cette choix trouve ses raisons dans le fait que la mutation structurelle apporte des modifications violentes sur l'individu qui ne donnent pas très souvent de bons résultats ; donc il est nécessaire d'avoir une technique *élitiste* pour garder les améliorations que l'on a obtenues jusqu'à la génération précédente.

Mutations

On a deux opérateurs de mutation : l'un sert pour une simple mutation des paramètres (poids synaptique, biais et betas), l'autre sert pour la mutation de la topologie.

La **mutation des paramètres** est faite en suivant la stratégie de l'auto-adaptation : à chaque paramètre, qui est une valeur réelle, est associé une sigma. On fait chaque fois la mutation de tous les paramètres, ce qui peut sembler exagéré en considérant que leur signification est très différent, mais l'expérience nous a montré des bons résultats. On a quand même donné des bornes pour la variation des betas, tandis que celle des poids et des biais n'a pas été bornée.

La **mutation structurelle** nous permet de varier la topologie ; il y a quatre façon de le faire : ajouter une connexion ou un neurone ou les enlever. Le réseau peut donc grandir ou rapetisser en s'adaptant à la meilleure structure. Toutes les mutations sont faites aléatoirement, sans suivre aucune heuristique : de cette façon on ne se limite pas au problème particulier que l'on est en train de résoudre, ce qui s'avère très important pour des problèmes d'optimisation compliqués.

- Pour rajouter une connexion on choisit un neurone au hasard dans l'ensemble des neurones d'entrée et cachés : c'est le neurone de départ de la connexion ; après on choisit un neurone-destination de façon que la connexion soit légitime ; le poids de la nouvelle connexion est mis a zéro afin que le comportement du réseau ne change pas et qu'elle aie la même performance qu'avant.
- Pour rajouter un neurone on choisit au hasard sa position dans la liste ; il se trouvera toujours parmi les neurones cachés puisque l'on ne peut pas changer le nombre des entrées ou des sorties. De plus, on rajoute deux connexions avec un poids nul, l'une qui a comme destination le nouveau neurone et l'autre qui l'a comme départ. Ce changement de topologie est

assez fort mais tout d'abord la performance reste la même et ceci nous permet de nous améliorer.

- Pour enlever une connexion on la choisit au hasard : dans ce cas on ne peut pas garder la même valeur de fitness, car on peut pas avoir un réseau équivalent. En fait, pour avoir moins de problèmes on aurait pu décider, de façon plus déterministe, d'enlever la connexion avec le poids le plus bas ; toutefois on a choisi de ne pas le faire parce que l'algorithme de la mutation doit être le moins complexe possible, et ce ne serait pas le cas avec un tel choix.
- Pour enlever un neurone on le choisit au hasard dans la liste : tout d'abord, pour la même raison qu'avant, on avait pensé à n'enlever que le neurones sans sorties, mais c'était un cas trop particulier qui ne se passait presque jamais ; donc on a décidé d'enlever un neurone quelconque, mais aussi de baisser au même temps la probabilité de faire une telle mutation.

Croisement

Trouver une façon de faire le croisement pour les réseaux de neurones n'est pas tout à fait évident, surtout si l'on veut garder le sens théorique de l'opérateur ; si l'on pense à des vecteurs de réels, il est naturel de penser de croiser l'information amenée par les parents en prenant une partie du premier et l'autre du deuxième ; mais avec deux réseaux qui de plus ont une topologie différente, quoi faire ? Le problème principal est représenté par la différence de topologie : en fait, si la structure était toujours égale, on pourrait simplement faire un croisement des paramètres comme si on avait un vecteur. Il est donc nécessaire d'avoir d'abord deux structures égales sur lesquelles on peut faire un croisement, pour enlever les redondances à la fin.

Le croisement que l'on a employé est fait sur deux réseaux considérés comme parents et il a trois phases (fig. 4.3) :

- **Complication** : on rajoute des neurones cachés et des connexions aux deux parents ; tous les paramètres de ce que l'on rajoute sont nuls. À la fin de ce processus on a deux réseaux avec le même nombre de neurones et totalement connectés, donc il s'agit d'une même topologie.
- **Croisement** : on construit le nouveau individu en prenant aléatoirement des neurones des parents, dont l'information est mélangée.

- **Simplification** : on enlève toutes le connexions et tous le neurones rajoutés, inutiles pour le calcul car ils ont de valeurs nulles qui ne peuvent pas changer (à cause de leur *sigmas* qui sont elles aussi nulles).

Ce procédé est général : si l'on part de la même topologie pour les deux parents, on arrive à un nouveau individu qui a encore la même topologie, mais dont les paramètres sont le croisement de ceux des parents, donc on a tout à fait un croisement des paramètres ; par contre, si au début les topologies sont différentes, on arrive à une nouvelle structure qui nous permet d'explorer des nouvelles possibilités.

Sélection

Comme on a utilisé les ESs, la sélection des individus qui forment la nouvelle population est déterministe, basée sur la performance de chacun ; par contre on a remarqué que le croisement à besoin d'un sélecteur différent de celui aléatoire qui est employé par défaut ; en fait, un tel croisement marche mieux si les réseaux sont assez différents.

Un opérateur de *distance* nous renseigne sur les différences entre deux réseaux par rapport à leur topologie et leurs paramètres. Pour la sélection du partenaire du croisement, on fait un 'tournoi' : on choisit q individus (q étant la *taille* du tournoi), on calcule la distance de chacun par rapport au premier parent et on choisit l'individu le plus différent. La taille q du tournoi détermine jusqu'à ce que le niveau de la sélection soit aléatoire : plus la taille est petite, plus le chiox est aléatoire).

Le Problème de la Convergence

Notamment les ESs sont connues pour leur caractéristique de *convergence*, puisqu'elles convergent l'optimum plus rapidement que d'autres techniques [1] ; toutefois, cela marche dans des cas assez simples et avec des bons opérateurs, autrement on risque de tomber dans un optimum local et de ne pas arriver à en sortir. La recherche de la meilleure topologie pour un réseau de neurones est difficile à résoudre, et la mutation ou le croisement donnent souvent des résultats assez décevants, puisqu'il produisent des changements trop fort dans l'individu. La sélection de la nouvelle population étant déterministe, favorise presque toujours des individus qui n'ont pas changé leur topologie : assez rapidement tous les individus dans la populations ont la même structure, et le processus d'optimisation a du mal à sortir de l'optimum local dans lequel toute la population est tombée. On peut trouver dans la littérature des différentes approches pour éviter ce problème [7][8], qui restent quand même limitées aux RMC.

On a pensé à réaliser une sorte de auto-adaptation du réseau après une mutation structurelle ou un croisement, afin que la nouvelle structure puisse régler ses paramètres et donc rivaliser avec les autres déjà partialement optimisées. Pour ce faire, après un changement quelconque de la topologie, on fait un boucle d'optimisation des seuls paramètres avec des simples mutations : si le nouveau réseau a des potentialités, on arrive à dépasser le meilleur individu, ou au pire, à entrer dans la prochaine population (en fait, on fait une $(1+1)$-ES)

4.4.3 Résultats

Tout d'abord on a essayé d'approcher des fonctions à deux entrées et une sortie, en employant comme fitness l'erreur carrée moyenne mesurée sur 100 points. Les premiers essais ont été faits sur des fonctions assez simples ; on a choisi la taille de la population $\mu = 30$ et donc $\lambda = 90$: cela nous permet d'avoir trois fils par parent. Tout d'abord on a travaillé sans croisement, et la seule mutation n'a pas donné des très bons résultats. En rajoutant le croisement, on a remarqué qu'il ne marche pas très bien au début, quand les topologies sont très différentes, mais qu'il marche mieux quand la topologie est déjà un peu optimisée, c'est-à-dire quand la population a commencé à converger : le croisement est plus performant si les réseaux ont quelques connexions différentes, mais pas beaucoup si elles ont un nombre de neurones différent ; cependant il ne faut pas généraliser cette observation puisque souvent on a un bonne réseau même de deux parents avec un nombre différent de neurones. Ce comportement du croisement nous a amené à développer une stratégie d'évolution qui fait partir l'évolution en faisant surtout des mutations structurelles et qui augmente la probabilité de croisement quand la population s'est améliorée.

Pour aider à résoudre le problème de la convergence (voir 4.4.2) on a décidé de baisser la taille des enfants générés : on est passé à une $(\mu + \mu)$-ES, avec $\mu = 30$: en fait, cela nous permet de sauvegarder la variance de topologie pour plus longtemps car chaque parent ne produit qu'un enfant.

On est arrivé à approcher des fonctions avec une erreur carrée moyenne de l'ordre de 10^{-5} en quelques centaines de générations. On peut remarquer le problème de la convergence et sa solution en regardant la figure 4.6. Elle représente l'évolution de la meilleure performance dans la population (en vert) et de la moyenne sur tous les individus (en rouge) : on peut remarquer qu'il y a une partie où la meilleure performance et la moyenne sont identiques : dans ce cas on a tous les individus qui ont la même fitness, donc une population qui a convergé vers une seule structure, le processus d'optimisation bloqué dans un optimum local ; heureusement la mutation structurelle a généré un nouvel individu qui, grâce à la boucle d'adaptation des paramètres (voir 4.4.2), rejoint une performance meilleure que l'optimum local trouvé et permet à l'algorithme d'optimisation de chercher un autre optimum.

4.4.4 Analyse de complexité

On va discuter brièvement de la complexité de l'algorithme évolutionnaire que l'on vient de décrire. Tout d'abord on introduit un peu de terminologie :

- μ, la taille de la population (dans notre cas aussi la taille de l'ensemble d'enfants générés par génération).
- n, le nombre maximum de neurones des réseaux .
- N, le nombre maximum d'évaluations dans le boucle pour l'optimisation des paramètre après une variation topologique.
- P_c, la probabilité de croisement ;
- P_{sm}, la probabilité d'avoir une mutation structurelle quelconque.
- $P_{\overline{sm}}$, la probabilité moyenne entre le quatre possibles mutations structurelles (voir 4.4.2).

On peut remarquer que la structure choisie a au maximum un nombre de connections proportionnelles à n^2. les opérations que l'on fait à chaque génération sont croisement, mutations et la "compilation" du réseau vers la structure optimisée pour le calcul. La complexité de chaque opérateur est :

- **Mutation des paramètre** : comme on doit parcourir tous les neurones et toutes les connections du réseau, la complexité est $C_m = n + n^2 \to O(n^2)$.
- **Mutation structurelle** : dans le cas le pire on doit parcourir toute la liste de neurones pour rejoindre l'endroit où faire la mutation ; la complexité est donc linéaire en n, $C_m s = O(n)$.

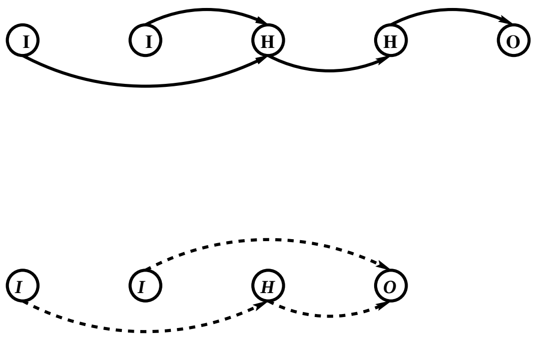
- **Croisement** : on a quatre phases : sélection du partenaire (calcul de la distance $\rightarrow O(n^2)$), complication des réseaux ($O(n^2)$), croisement des individus compliqués ($O(n^2)$) et finalement simplification de l'enfant ($O(n^2)$). Donc totalement on a une complexité de l'ordre de ($O(n^2)$).

La mutation structurelle et le croisement ont besoin du boucle d'optimisation de paramètres, donc à leur complexité il faut ajouter un terme qui en tien compte, de l'ordre de $N \cdot O(n^2)$. En employant un schéma d'évolution $(\mu + \mu)$-ES, chaque parent génère un enfant, et les opérateurs ne sont appliqués qu'à ce dernier, donc on a μ appels par chaque génération des opérateurs. En considérant le probabilité d'avoir une mutation ou un croisement la complexité globale est :

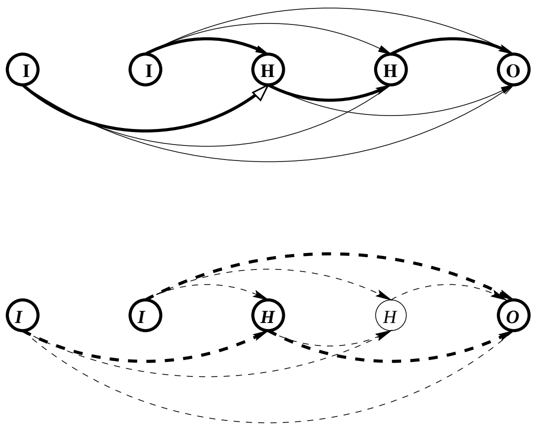
$$\begin{aligned} C(n, \mu) = \mu[P_c(O(n^2) + NO(n^2)) + P_{ms}P_{\overline{ms}}(O(n) + NO(n^2)) + \\ + (1 - (P_c + P_{ms}P_{\overline{ms}}))O(n^2)] \end{aligned} \tag{4.1}$$

Évidemment évoluer une génération a une complexité de l'ordre de $\mu \cdot O(n^2)$.

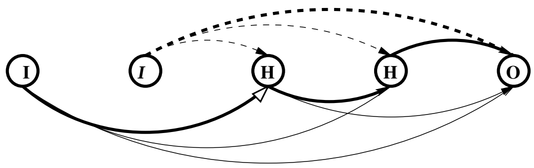
Il ne faut pas oublier la *fitness* : il est claire que d'elle dépend de comportement de l'entier algorithme : en fait chaque fois que l'on fait une mutation à l'individu on doit la calculer pour détecter des éventuelles améliorations, donc si on fait une approximation, à la limite on la calcule $N\mu$ fois. Dans le cas de l'approximation de fonctions la *fitness* est très simple et ne ralentit pas l'évolution de la population.



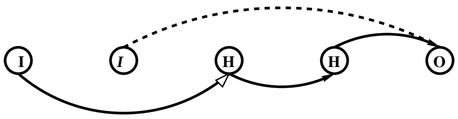
(a) départ



(b) complication



(c) croisement



(d) simplification

FIG. 4.3: Le croisement : (a) représente les parents avant de le croisement ; on peut observer que le deux topologie sont différentes. (b) représente les parents après la “complication”, c’est-à-dire ils ont la même topologie totalement connectée (les parties rajoutés étant moins épaisses). (c) représente l’individu après le croisement : on peut voir le neurones du premier parent (nom en texte roman), du deuxième(en *italique*), avec leur connexions. (d) représente l’individu après la simplification des connections de poids nul.

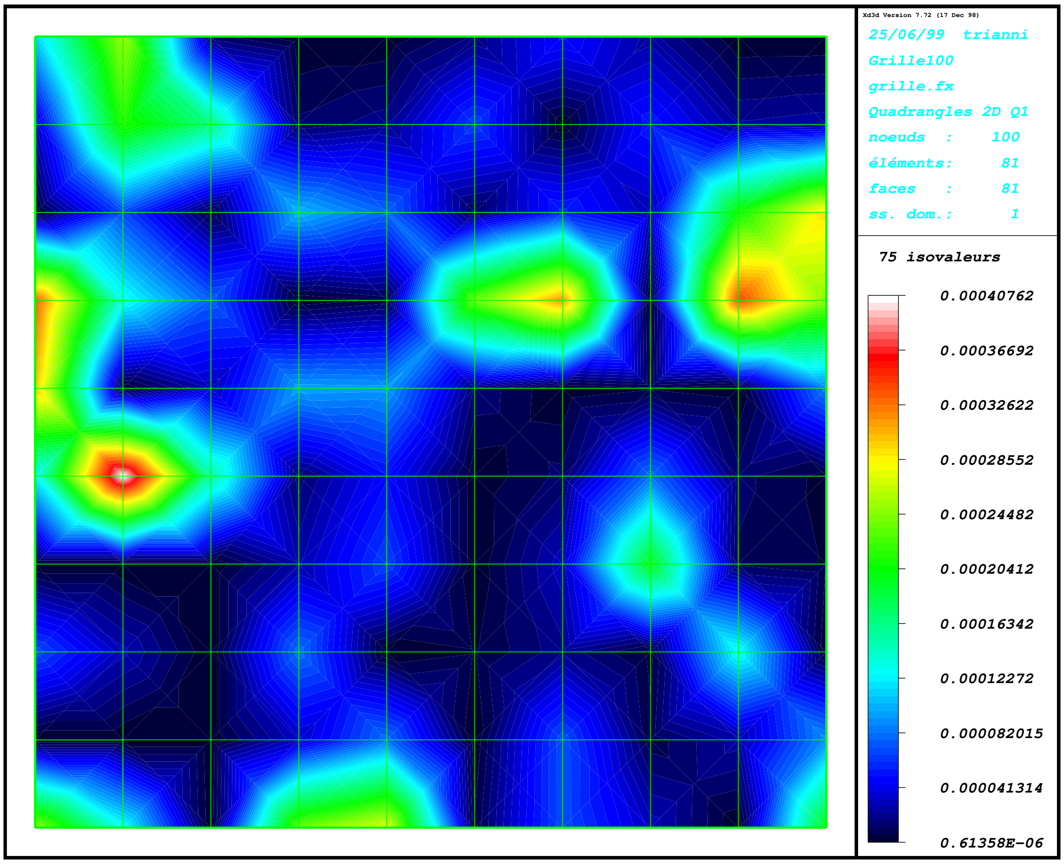
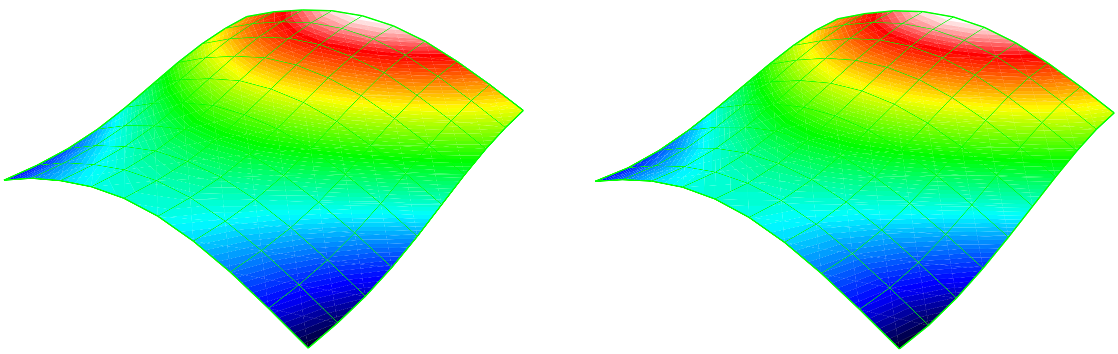


FIG. 4.4: l'erreur



(a) la fonction calculée par le réseau

(b) la fonction exacte

FIG. 4.5: les fonction calculée comme exemples a été $f(x,y) = \sin x - \cos y$

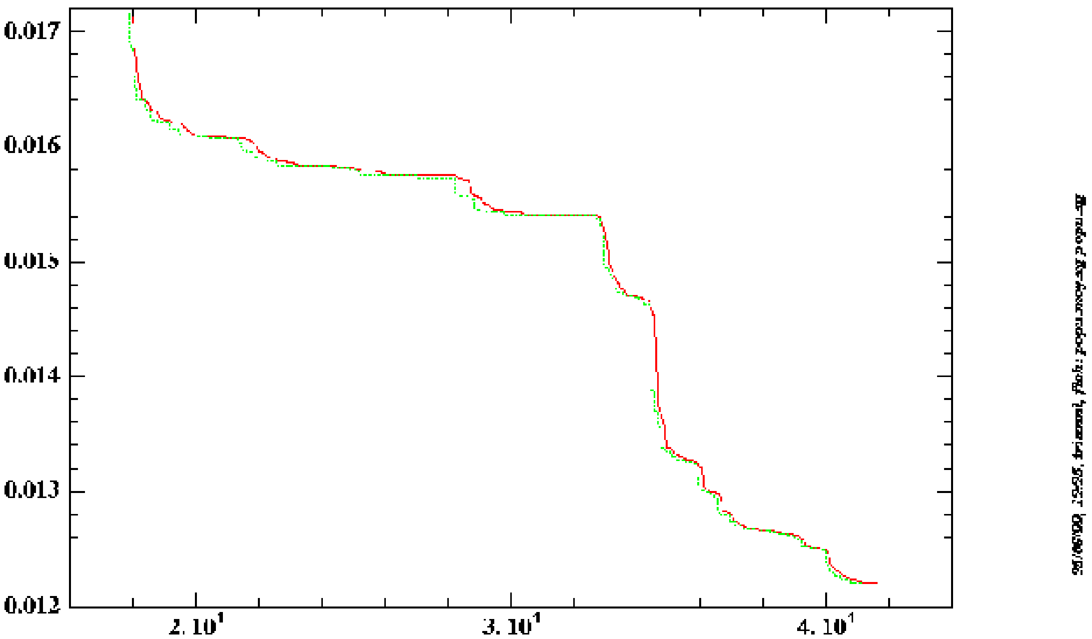


FIG. 4.6: le problème de la convergence : la meilleure performance (en vert) et la performance moyenne de la population (en rouge) ; on peut remarquer des optima locaux quand les deux ont la même valeur.

Chapitre 5

Expériences avec le Khepera

5.1 Introduction

Ce chapitre décrit les expériences que l'on a fait sur les simulateurs du robot Khepera pour lui apprendre à éviter les obstacles. On parle des premiers essais que l'on a faits sur le simulateur 2D et du passage au simulateur 3D (voir 3.2). On décrira en particulier le premier contrôleur que l'on a développé, l'apprentissage d'un contrôleur par Algorithmes Évolutionnaires, l'apprentissage d'un contrôleur avec des réseaux de neurones qui nous a mené à l'expérience "Recherche de la nourriture".

5.2 Le simulateur 2D

Dans cette section on parle du travail que l'on a fait sur le simulateur 2D pour se familiariser avec le Khepera.

5.2.1 Le contrôleur "statique"

Dans cet environnement on a développé un simple contrôleur pour le robot dont l'objectif est éviter les obstacles. Pour faire ceci on s'est inspirés à la méthode des potentiels que l'on va décrire rapidement. On peut penser aux obstacles et au robot comme deux charges du même signe qui se repoussent, la répulsion étant plus forte quand le robot et l'obstacle sont plus proches ; cela nous donne un contrôleur qui garde le robot loin des obstacles.
Pour implémenter la répulsion il faut que chaque capteur donne une contribution

à la vitesse des deux moteurs. Comme la configuration des capteurs n'est pas symétrique il faut la rendre telle en pesant la valeur donnée par eux. Ce contrôleur a donné des bons résultats dans un environnement simple, c'est-à-dire avec peu d'obstacles mais, dans un environnement plus compliqué (comme un labyrinthe) il a eu des difficultés à déplacer le robot, dans les couloirs étroits, par example.

5.2.2 Apprentissage d'un contrôleur par Algorithmes Évolutionnaires

Pour dépasser les problèmes rencontrés avec le contrôleur duquel on vient de parler, on a essayé d'apprendre les paramètres à utiliser comme poids des capteurs. Pour l'apprentissage on a employé EvolC (voir 1.2) : l'individu est un vecteur de réels qui code les paramètres du contrôleur ; on utilise les stratégies d'évolution (ESs) avec mutation adaptative et croisement standard (voir 1.1.2). Avec ce contrôleur on a vu que le robot, au moyen de l'apprentissage, peut bouger dans différents environnements, mais il y a des situations dans lesquelles il a des problèmes (par exemple il reste parfois bloqué sur un coin), c'est-à-dire qu'il n'arrive pas à les apprendre. On peut donc dire que ce contrôleur est sans doute meilleur que celui sans apprentissage, mais il ne peut pas résoudre au mieux le problème d'éviter les obstacles. Il faut dire que quand on a développé ce contrôleur on n'avait pas encore trouvé une bonne fonction de fitness (voir 5.4.1) et donc peut-être que les résultats que l'on peut obtenir avec ce contrôleur soient meilleurs.

5.3 Le simulateur 3D

On a décidé d'employer Webots au lieu de la première version du simulateur pour avoir moins de problèmes d'interface avec le programme d'évolution (avec la première version il fallait mettre ensemble simulateur et programme d'évolution, tandis que avec Webots on peut utiliser la technique du superviseur (voir 3.2)), et surtout pour avoir la possibilité de profiter de toutes les capacités nouvelles que l'on a dans Webots ; toutefois on a perdu en vitesse de calcul (simulation 3D), surtout après avoir rajouté la caméra au robot.

5.4 Un Controlleur Neuronal

L'apprentissage du réseau comme contrôleur du Khepera est fait en employant le simulateur Webots (voir 3.2) ; on a utilisé la même implémentation que pour l'approximation des fonctions : l'individu est un réseau qui a comme entrées les valeurs des capteurs du Khepera et comme sortie les vitesses des moteurs ; ce qui change est bien sûr la fitness, qui doit être calculée selon le comportement du robot. Afin de vérifier la possibilité de transférer vers un robot réel des contrôleurs neuronaux issus du processus évolutionniste, nous avons mis au point une expérience qui permet d'obtenir des contrôleurs neuronaux dotant un robot d'un comportement de navigation avec évitement d'obstacles.

5.4.1 Fitness

Une fois que l'on a décidé sur comment apprendre un réseau, la fitness est le paramètre le plus important à définir, car elle change de problème en problème : pour éviter les obstacles, on veut que le comportement du robot lui permet de ne pas se coincer et d'explorer l'environnement le plus efficacement possible.

Tout d'abord on peut remarquer qu'il y a plusieurs grandeurs que l'on peut maximiser :

- **Distance parcourue**, car on veut que le robot bouge le plus possible.
- **Distance euclidienne**, car on veut que le robot s'éloigne le plus possible de la position initiale.
- **Vitesse**, mesurée comme la somme des vitesse des moteurs, car on veut que le robot aille le plus rapidement possible et surtout droit (le deux moteurs à la vitesse maximale donnent une trajectoire droite).
- **Le nombre des carreaux** visités, un carreau étant une section carré de l'environnement

Le calcul de la fitness se fait en donnant au début un nombre fixé de pas de simulation au robot et en calculant le grandeurs a maximiser. On peut donc les combiner pour obtenir une valeur unique de fitness. Chaque grandeur a ses avantages et ses désavantages, et aucune toute seule suffit pour donner le meilleur comportement.

Distance Parcourue

La distance parcourue est la première chose à laquelle on pense, parce que on veut que le robot marche dans l'environnement sans se coincer : si on maximise la distance parcourue on sait que l'individu qui évite les obstacles obtient la fitness la plus haute. Par contre le comportement que l'on obtient est simultanément bizarre et raisonnable : dans une environnement sans obstacle le robot parcourt des cercles, tandis qu'en face d'un obstacle il avance et recule à la vitesse maximale, ce qui lui permet de bouger plus ! Ce comportement se manifeste à cause de la manque du concept d'inertie dans le simulateur.

Distance Euclidienne

La distance euclidienne est pensée pour faire éloigner le robot le plus possible de la position initiale. Le seul problème est que, dans un environnement plein d'obstacles même un bon contrôleur qui fait bouger le robot très bien a une performance très basse, parce que le robot a choisi de retourner sur ses pas.

Vitesse

Maximiser la vitesse a deux effets : premièrement le robot est plus rapide, deuxièmement il va tout droit jusqu'à qu'il soit possible ; le problème arrive ici aussi quand l'environnement est plein d'obstacles, parce que, comme pour la distance euclidienne, n'avoir pas une vitesse maximale n'implique pas avoir une mauvaise performance.

Carreaux Parcourus

L'idée est de donner un score au robot pour chaque carreau que le robot traverse pour la première fois : en le maximisant, on obtient un comportement qui amène le robot à explorer tout l'environnement ; la façon d'utiliser cette grandeur est un peu différente : on donne au robot un nombre de pas de simulation pour chaque carreau traversé, donc la durée de la simulation dépend du comportement du robot.

On peut remarquer que pour obtenir le meilleur comportement il faut utiliser plusieurs grandeurs ensemble, afin de profiter au maximum des avantages et d'annuler les désavantages. La meilleure solution a été de calculer la performance comme $Vitesse{+}Carreau\ Parcouru$, car le couple peut même maximiser la distance parcourue.

5.4.2 Apprentissage

On a fait évoluer une population de $\mu = 20$ individus, avec un schéma $(\mu + \mu)$-ES ; chaque calcul de fitness se passe dans un labyrinthe : toute la population doit faire le même calcul afin que les fitness soient comparables ; il s'avère qu'il y a des positions de départ simples et d'autres plus difficiles : l'apprentissage doit être graduel, en commençant par de cas faciles pour les compliquer une fois que l'on les a appris. Cela permet de ne pas se spécialiser pour des cas particuliers, mais d'être prêt pour une situation quelconque.

5.4.3 La Symétrie

Le réseau de neurones qui doit donner le contrôleur au robot prend comme entrées les valeurs des capteurs de proximité et calcule les valeurs de vitesse des moteurs. Tout d'abord on a fait évoluer les individus qui ont comme entrées les huit capteurs et qui calculent les vitesses des moteurs comme sorties ; le problème est assez difficile parce qu'il faut apprendre pour chaque combinaison d'entrées les meilleures sorties : les expériences ont montré que le robot arrive à éviter les obstacles, mais il se spécialise, c'est-à-dire il n'apprend qu'à tourner soit à gauche, soit à droite, tandis qu'il faudrait avoir un comportement symétrique, le robot étant symétrique. Une possibilité serait d'ajouter à l'apprentissage un *opérateur de symétrie* qui prendrait un réseau et la rendrait symétrique ; un tel opérateur est difficile à trouver : on peut avoir le réseau symétrique d'un réseau donné en changeant les neurones d'entrée et de sortie avec leurs correspondants, mais ceci ne nous donne qu'une autre solution partielle.

On a forcé le comportement à être symétrique en changeant l'individu : on a toujours un seul réseau qui contrôle le robot, mais il ne contrôle qu'une roue à la fois en regardant la moitié des capteurs (fig. 5.1). Dans ce cas, le problème est très simplifié, mais la solution est tout à fait générale car on souhaite toujours un comportement symétrique.

5.4.4 Résultats

L'évolution a généré des contrôleurs qui ont rendu le robot capable d'éviter les obstacles, même dans des environnements complexes comme un labyrinthe. Les

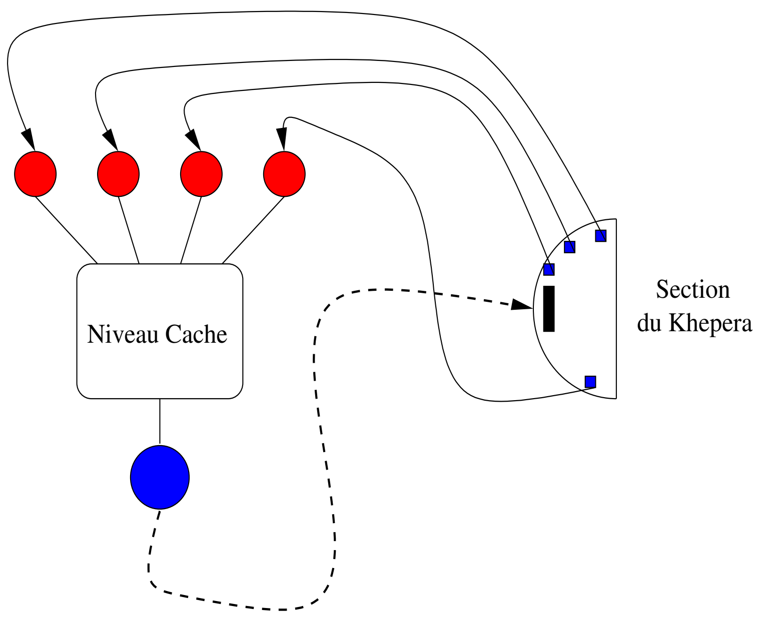


FIG. 5.1: Schéma des entrées et des sorties

problèmes qu'il y avait avec les contrôleurs inspiré à la méthode des potentiels ont été complètement réglés, car le réseau peut traiter mieux toute configuration des entrées donnée par les capteurs.

La figure 5.2 montre l'évolution de la meilleure performance pendant l'apprentissage dans un labyrinthe. On peut remarquer qu'au début la performance atteint une valeur assez haute dans peu de générations partant d'un cas simple : le robot peut bouger partout sauf que dans de couloir très étroits ; puis l'apprentissage continue jusqu'à rejoindre une performance optimale : le robot a amélioré son comportement aussi dans les endroit plus étroits.

Dans le graphique on peut voir des points où la performance diminue : il s'agit du changement des points de départ dans l'environnement ; quand on passe á un cas plus difficile, souvent la performance de tous les individus tombe, parce que ils se trouvent en face d'un problème plus difficile que il n'ont encore appris à résoudre. Cette technique d'avancer petit à petit nous permet de résoudre le problème de façon graduelle et de atteindre l'optimum plus rapidement.

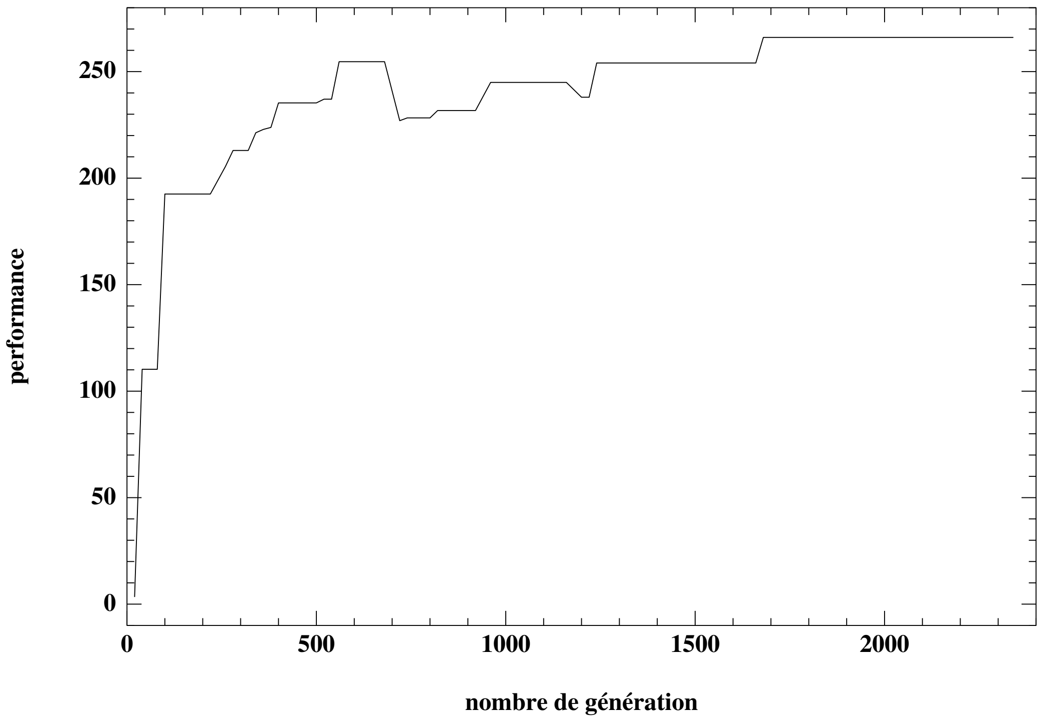


FIG. 5.2: L'évolution de la performance dans un labyrinthe

5.5 Recherche de nourriture

5.5.1 Introduction

"The Artificilal Life Creator Contest" est une compétition entre deux robots : il sont programmé pour survivre dans une environnement qui devient toujours plus hostile : ils doivent chercher des "distributeur de nourriture" (*feeders*) qui leur donnent l'énergie pour continuer à chercher. Les *feeders* se rechargent toujours plus lentement, donc il devient difficile pour les deux robots de survivre. Le gagnant est le robot qui vit plus longtemps.

Dans le but de vérifier l'efficacité de nos réseaux de neurones et poussés par le concours, nous avons mis au point une expérience qui a pour objectif de faire émerger des contrôleurs neuronaux qui soient capables de piloter un robot mobile dans un environnement pour l'explorer á la recherche des *feeders*.

5.5.2 Environnement

On a réalisé l'apprentissage du réseau dans un environnement qui contient des obstacles et des *feeders* repartis assez uniformément (fig. 5.3) et mis en positions plus ou moins faciles å trouver par le robot. Les *feeders* sont des parallélépipèdes caractérisés par un rectangle vert sur une de leurs faces, qui indique la présence de la "nourriture" (fig. 5.4).

FIG. 5.3: L'environnement

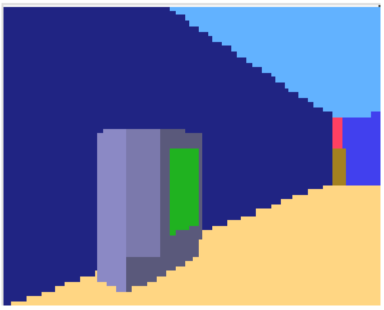


FIG. 5.4: Un *feeder*

La robot les perçoit grâce à la caméra. Afin de permettre au robot de ne pas confondre les *feeders* avec les autres obstacles, on ne utilise la couleur vert que pour la "nourriture". Ce que l'on vient de décrire est l'environnement dans lequel se fait l'entraînement du robot. L'environnement du concours est tout-á-fait identique, sauf qu'il y a deux robot qui se "battent" pour la "nourriture".

5.5.3 Mise en place des expériences

Pour essayer de résoudre cette problème on à gardé la même implémentation de réseau de neurones que l'on a employée dans l'expérience pour éviter les obstacles, en ajoutant des entrées pour traiter les données envoyées par la caméra afin de détecter les *feeders*. L'image prise par la caméra est filtrée pour permettre au robot de ne voir que le vert ; puis l'image filtrée est divisée en quatre bandes et on utilise comme information à donner au robot la quantité de pixels verts présente dans chaque section de l'image : cela nous donne environ la position du *feeder* et aussi sa distance. On ne fait pas une filtrage de l'image qui nous permet de reconnaître les formes car on espère que l'apprentissage arrive à distinguer les différentes situations. Même pour ce problème on a gardé la structure symétrique pour le réseau (voir 5.4.3), qui dans ce cas est caractérisée par six entrées : quatre pour les valeurs données par les capteurs de proximité et deux pour celles données de la caméra (fig. 5.5).

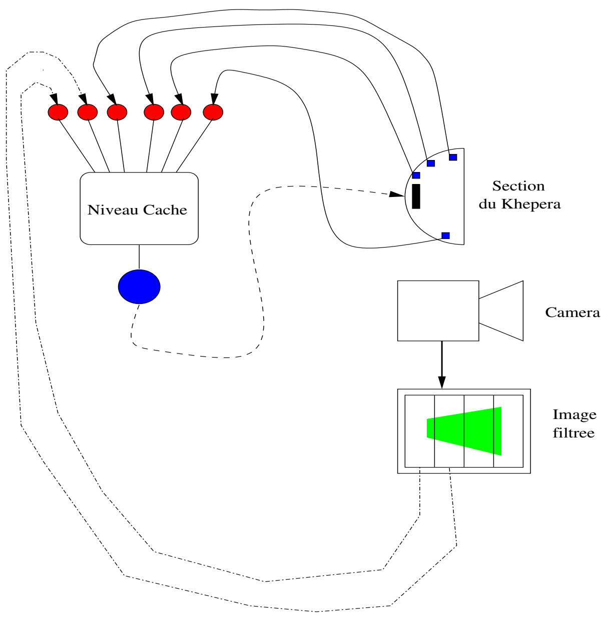


FIG. 5.5: Schéma des entrées et des sorties avec caméra

Pour ce qui concerne la fonction de fitness on l'a dérivée de celle que l'on avait choisi pour éviter les obstacles (voir 5.4.1) en combinant les paramètres suivants :

- vitesse des roues.
- nombre de carreaux visités.
- nombre de pixels vertes vus dans un pas de temps multiplié pour -1 si le deux roues ont tout le deux une vitesse négative : cela nous permet d'obtenir un comportement qui fait marcher le robot vers le feeder.

On a décidé d'utiliser une fitness qui nous permet d'éviter les obstacles car évidemment il ne faut pas que le robot se coince ; de plus le nombre de carreaux visité permet d'optimiser un comportement qui explore l'environnement. Pour ce qui concerne la choix d'enlever des point à la fitness si le robot marche en arrière, on l'a fait pour pénaliser un comportement qui fait osciller le robot en face du *feeder* et pour lui donner "envie" d'y aller.

L'évolution des individus a été fait en deux phases : dans la première le robot apprend à bouger dans l'environnement (pas forcement le même) sans avoir des information sur une présence éventuelle de "nourriture" (le nombre de pixels verts détectés est toujours mis à zéro) ; après on rajoute les informations sur le vert pour lui apprendre à chercher les *feeders*. On a essayé plusieurs approches pour l'apprentissage, entre lesquelles deux sont les plus intéressantes : l'adaptation de la topologie entière et l'adaptation de poids en ayant des structures partiellement optimisées pour éviter les obstacles et á notre avis capables d'atteindre la tâche "recherche de la nourriture". Le problème que nous a amené à essayer de telles approches est le temps-calcul nécessaire pour la simulation : comme on rajoute la caméra, la vitesse du simulateur tombe de 200 fois ! Apprendre la topologie de telle façon prend trop de temps, surtout a cause de le boucle d'optimisation de les poids pour l'adaptation d'une topologie muté. Par contre en ayant une topologie déjà assez complexe et en n'optimisant que ses paramètres on espère d'obtenir un bon résultat en moins de temps.

5.5.4 Résultats

On a obtenu les meilleures résultats par la deuxième approche : en fait le robot explore l'environnement et quand il voix le *feeder* il cherche de le rejoindre ; la façon d'explorer l'environnement est différent par les expérience précédentes : le robot, pendent qu'il marche tout droite bouge un peu à droite et à gauche pour voir avec la caméra une partie plus grande de l'environnement : cela dépend du fait que les capteurs sont bruités, donc même quand il n'y a pas d'obstacles leur valeurs peuvent être non nulles ; le robot a appris donc à exploiter les bruits pour faire une sorte de *marche aléatoire* dans l'environnement, qui est plus performante d'une simple comportement qui évite les obstacles.

Comme on a déjà remarqué (voir 4.4.4), le calcul de la fitness se fait beaucoup de fois pendant une seule génération : l'expérience avec le robot nous a donné des fonctions de performance qui, à cause du simulateur, ont ralenti l'évolution sans nous permettre d'avoir les résultats que l'on a obtenus en temps raisonnable. Cela nous démontre que dans certains cas le calcul de la performance peut devenir un paramètre critique pour la marche de l'évolution.

Chapitre 6

Conclusions

Les algorithmes évolutionnaires offrent la possibilité d'optimiser des problèmes complexes surtout quand on n'a pas d'exemples ; pour faire évoluer la topologie d'un réseau de neurones on a besoin d'un instrument qui soit puissant et versatile, pour nous permettre de s'adapter à chaque problème que l'on peut envisager. Les stratégies d'évolution avec des opérateurs qui permet à l'individu de s'adapter le plus possible à le problème particulier que l'on est en train de résoudre ont donné de très bons résultat pour de problèmes simples comme l'approximation de fonctions, mais aussi pour des problèmes plus complexes comme l'apprendre un contrôleur pour un robot qui doit atteindre une tâche donné. Ce type d'apprentissage se montre assez général et assez promettent pour une extension à la résolution de problèmes de plus en plus complexes.

Annexe A : structure des classes

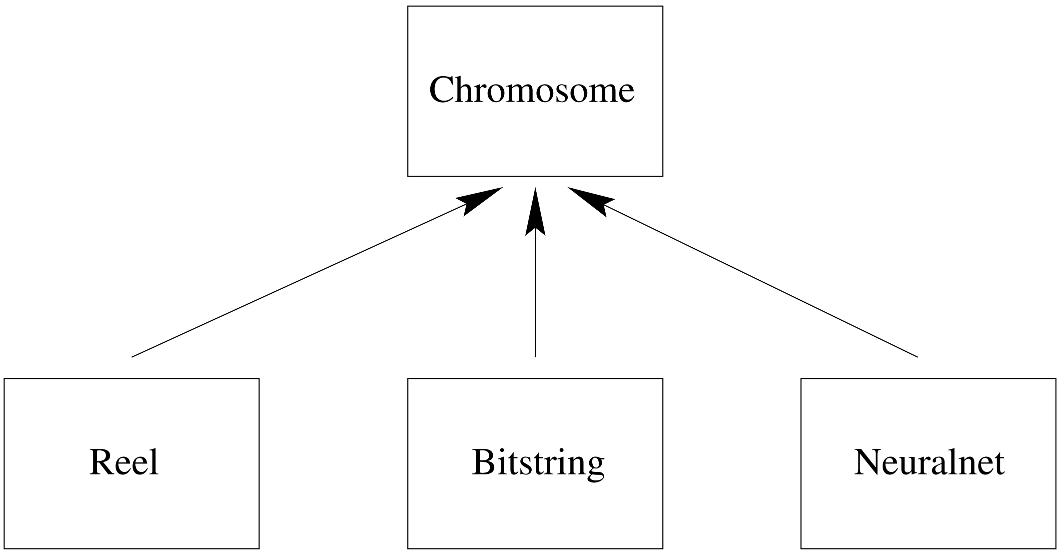


FIG. 6.1: Schéma des classes qui représente les individus ; une classe de base, *Chromosome*, fournit tous les définitions des méthodes qui doivent être employées pendent l'évolution (*mutation*, *croisement*, *clonation*, *stratégies*). Des sous-classes définit cettes méthodes pour le différentes génotypes ; on la classe Réel dont le génotype sont des vecteurs de réels, la classe Bitstring pour les vecteurs de bits ; ceux deux classes sont standard mais on peut l'adapter pour les différentes nécessites ; de plus on peut définir de nouveaux génotypes, comme nous avons fait pour les réseaux de neurones.

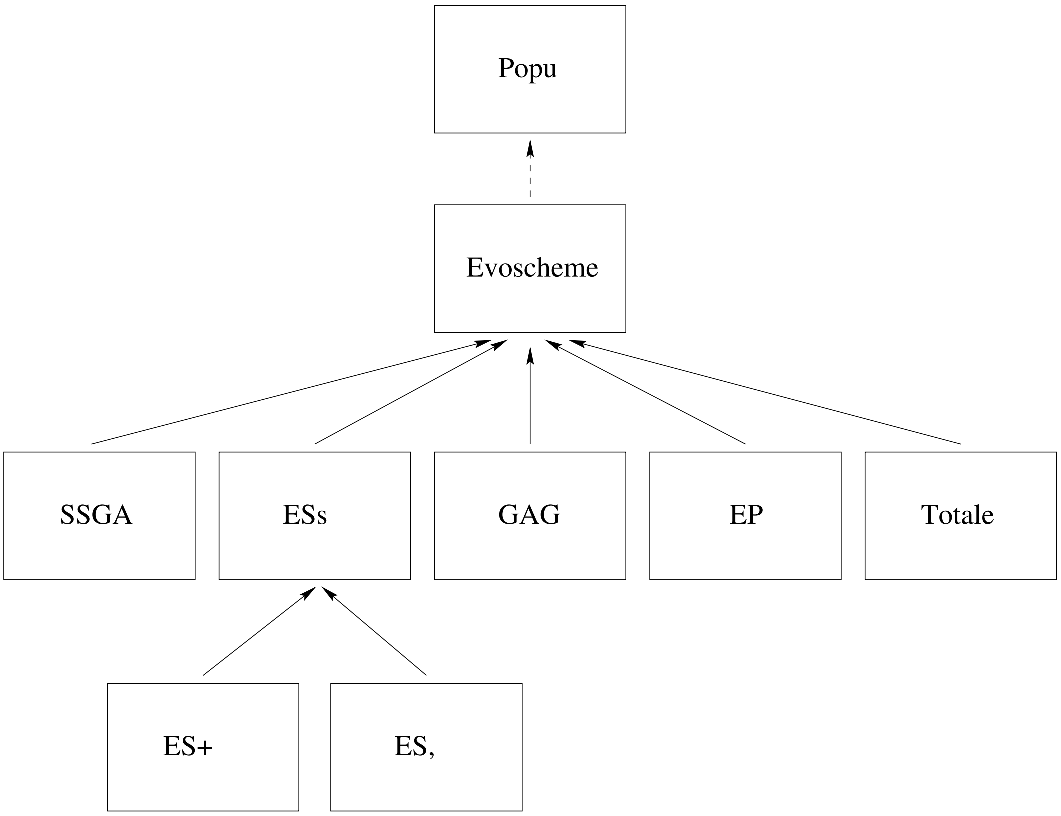


FIG. 6.2: Schéma des classes : la classe Popu gère le fonctionnalité de la population : c'est ici que l'on contrôle les processus d'évolution en emploient une schéma d'évolution parmi ceux définis par la classe Evoscheme et ses classes derivees : *Steady State Genetic Algorithm* (SSGA), *General Algorithme Génétique* (GAG), *Strategie d'évolution* (ES dont on dérive ES+ et ES,), *Evolutionary Programming* (EP)

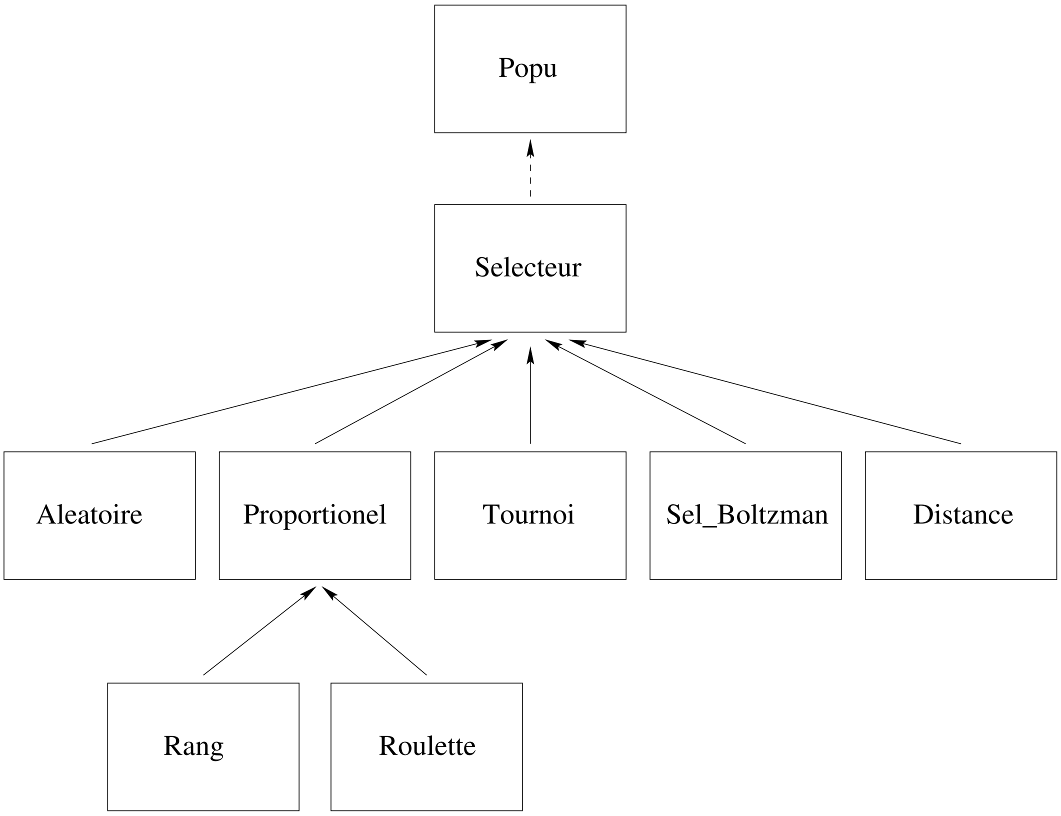


Fig. 6.3: Schéma des classes : la classe Popu gère aussi la sélection des individus pour construire la nouvelle population en partant de celle courante ; pour le faire il y a la classe Sélecteur qui est spécifie par les différent type de sélecteurs possibles : sélection *aléatoire*, sélection *proportionelle*, *tournoi*, *sélecteur de boltzmann*, *distance*, qui nous avons définit pour le croisement entre deux réseaux.

Bibliographie

[1] Thomas Bäck, Hans-Paul Schwefel. An overview of Evolutionary Algorithms for Parameter Optimisation, *Evolutionary Computation*, 1(1) :1-23, 1993.

[2] Thomas Bäck. Evolution Strategies : An Alternative Evoltionary Algorithm.

[3] Marc Shoenauer, Zbigniew Michalewicz. Evolutionary Computation, *Control and Cybernetics*, 26(3) pp 307-338

[4] Floreano, D. and Mondada, F. Evolution of Homing Navigation in a Real Mobile Robot. *IEEE Transactions on Systems, Man and Cybernetics Part B : Cybernetics*. 26, 3, 396-407.

[5] Jakobi, N. Running Aross the Reality Gap : Octopod Locomotion Evolved in a Minimal Simulation, *Lecture Notes in Computer Science 1468, Evolutionary Robotics*. 39-58.

[6] Meyer, J.A. Husbands, P.H. Harvey, I. Evolutionary Robotics, A Survey of Applications and Problems

[7] Moriarty, D.E. Miikkulainen, R. Forming Neural Networks through Efficient and Adaptative Coevolution, *Evolutionary Computation*, 5(4), 1998.

[8] Golubski, W. Feuring, T. Evolving Neural Network Structures by Means of Genetics Programming.

[9] Radi, A. Poli, R. Genetic Programming Discovers Efficent Learning Rules for the Hidden and Output Layers of Feedfoward Neural Networks.