SAMOTH[1]
Rapport intermédiaire

Samy ARAFA Clément CANONNE Romain GUION
Thibaut MÉTIVET Eliott MONNET Gautier VIAUD

30 novembre 2010

1. Encadrants : Nikos PARAGIOS, Loïc SIMON et Olivier TEBOUL (Laboratoire de Mathématiques Appliquées aux Systèmes (MAS), École Centrale Paris).

Table des matières

1 Introduction **3**
1.1 Qu'entend-on exactement par « intelligence artificielle » ? 3
1.2 Contexte 4
1.2.1 Jeux vidéo 4
1.2.2 Intelligence artificielle 4
1.3 Problème, formulation et objectif 5

2 Quelle intelligence artificielle et algorithmes sont actuellement utilisés dans les jeux vidéos ? **6**
2.1 *Scripting* vs. IA 6
2.2 Exploration d'arbres et algorithmes de recherche 7
2.2.1 Généralités 7
2.2.2 Heuristiques, A* 8
2.2.3 Min-Max, Alpha-Beta 9
2.2.4 UCB et UCT 13
2.3 Dans les jeux « classiques » (échecs, go . . .) 15
2.4 Dans les jeux de stratégie en temps réel 19
2.5 Dans les jeux de tir à la première personne 20
2.6 Et ailleurs (jeux hors catégorie) 21

3 État de l'art des utilisations de *réelle* intelligence artificielle **23**
3.1 Généralités 23
3.1.1 Concepts généraux 23
3.1.2 Agents rationnels 25
3.2 Réseaux de neurones 27
3.2.1 Description et formalisation mathématique 27
3.2.2 Processus d'apprentissage 30
3.3 Algorithmes génétiques 34
3.3.1 Une analogie avec la biologie 34
3.3.2 Mise en œuvre 35
3.3.3 Applications actuelles 36
3.4 Apprentissage par renforcement 37
3.4.1 Principe 37
3.4.2 Apprentissage par renforcement passif 38
3.4.3 Apprentissage par renforcement actif 41

4 Notre projet : spécificités et notions théorique nécessaires à sa bonne poursuite **44**
4.1 Spécificités du projet et conséquences 44
4.1.1 Les besoins d'un jeu vidéo en termes d'intelligence artificielle 44
4.1.2 Premier pas vers le jeu : un prototype . . . non jouable . . 45
4.2 Utilisations potentielles des réseaux de neurones 46
4.3 Pistes d'intégration des algorithmes génétiques 47
4.4 Apprentissage par renforcement 48
4.5 Agents et protocole 50

5 Conclusion **52**

Bibliographie **53**

Chapitre 1

Introduction

> « Parmi toutes les variétés de l'intelligence découvertes jusqu'à présent, l'instinct est, de toutes, la plus intelligente. »
> (Friedrich Nietzsche, *Par-delà le bien et le mal*)

1.1 Qu'entend-on exactement par « intelligence artificielle » ?

Le terme « intelligence artificielle » [1] (IA), bien que fréquemment utilisé et désormais entré dans le langage courant, demeure relativement flou et sujet à différentes interprétations - et ce, même sans tenir compte de son usage dans la littérature et science-fiction en général.

Ainsi, l'IA est parfois définie comme « la recherche de moyens susceptibles de doter les systèmes informatiques de capacités intellectuelles comparables à celles des êtres humains » [2], ou comme « l'étude et la conception d'*agents rationnels* », c'est-à-dire de systèmes qui, en fonction de ce qu'ils perçoivent de leur environnement, agissent de manière à maximiser leurs chances de succès [RN03] ; ou encore, et c'est par exemple le principe du test de Turing, comme la capacité à *simuler* un comportement humain.

Ces divergences de définition recouvrent en réalité une différence d'interprétation quant au *but* recherché - ce qui revient à distinguer quatre axes d'étude possibles :

- des systèmes qui *agissent* comme les êtres humains
- des systèmes qui *pensent* comme les êtres humains
- des systèmes qui *pensent* rationnellement
- des systèmes qui *agissent* rationnellement

Les deux premiers cas se réfèrent de facto à une certaine idée de l'Homme, d'où la nécessité de passer par l'étude de son esprit ou comportement - rien ne présuppose en effet a priori que les décisions prises ou actions effectuées par un programme suivant l'un de ces deux objectifs soient les *meilleures*, ou les plus *logiques*. Les deux derniers objectifs, en revanche, renvoient à une conception de

1. L'expression « intelligence artificielle » date de 1956, et est due à John McCarthy, inventeur du langage LISP et professeur émérite à l'Université de Stanford.
2. *La Recherche*, janv. 1979, n ° 96, vol. 10, p. 61

l'intelligence en tant que *raison*, et dégagée de toute considération psychologique ou sociologique. Enfin, la distinction effectuée entre *penser* et *agir* réside dans le point de vue, intérieur ou extérieur - selon que l'on considère la décision qui sous-tend les actes, ou uniquement le résultat de ces derniers.

Dans le cadre de notre projet, nous avons choisi de nous intéresser au dernier de ces objectifs, à savoir celui des *agents rationnels*. En effet, bien que chacun des quatre axes présentés soit également cohérent et justifiable, nous cherchons pour notre part à obtenir des comportements **objectivement évaluables** - d'où une approche rationnelle, jugés depuis l'extérieur. Pour le formuler différemment, notre étude et projet visent à créer des entités « en situation » **agissant de la manière la plus efficace possible** - et ce, sans préjuger du caractère "bon" ou "mauvais" des processus décisionnels à l'origine de leurs actions.

1.2 Contexte

1.2.1 Jeux vidéo

L'*industrie du jeu vidéo* est relativement jeune : une trentaine d'années [3] - cependant, elle représente aujourd'hui un marché mondial non négligeable (environ 30 milliards d'euros en 2003, avec un taux de croissance annuel de 15% en moyenne [4]), générant plus de 32000 emplois directs aux États-Unis, où son taux de croissance annuel a été en moyenne de 10,6% entre 2005 et 2009, contre 1,4% de croissance globale. [5].

Le développement de jeux vidéo nécessite désormais des équipes de plusieurs dizaines de personnes, chacune spécialisée dans un domaine particulier : programmeur, scénariste, *level designer*, modeleur 3D, concepteur d'interface utilisateur, *debugger* ... En parallèle de la professionnalisation du secteur, les joueurs deviennent eux-mêmes de plus en plus exigeants, et le réalisme (tant du point de vue graphique que des interactions avec l'environnement et des réactions des personnages non joueurs, contrôlés par l'ordinateur (ou *PNJ*)) devient un point crucial. Car même s'il est excellent sur d'autres points, un jeu prévisible dont les entités ne changent jamais de comportement, doté d'une IA basique et offrant peu d'occasions au joueur de sortir des sentiers battus risque très vite - sauf cas particuliers [6] - de se révéler répétitif et lassant, et donc décevant.

1.2.2 Intelligence artificielle

Après avoir connu un formidable enthousiasme à ses débuts, dans les années 50 et 60, puis une désaffection et une perte d'intérêt brutales dans la décennie suivante, lorsque les résultats obtenus ne se révélèrent pas à la hauteur des énormes espoirs qu'elle avait suscités, l'intelligence artificielle connaît de nouveau un essor rapide, en particulier grâce à l'évolution fulgurante des moyens

3. Le premier jeu vidéo, *Pong*, date de 1972 ; mais ce 'est qu'environ 10 ans après qu'une véritable industrie s'est mise en place.

4. D'après un rapport gouvernemental (français) daté de 2003 : `www.industrie.gouv.fr/pdf/rapportjeuvideo.pdf`

5. *Video Games in the 21st Century : The 2010 report*, Stephen E. Siwek, ESA : `http://www.theesa.com/`

6. eg, Tetris - le jeu simple, répétitif et néanmoins indéniablement populaire par excellence.

informatiques disponibles. Devenue une **industrie** dans les années 1980, avec par exemple les systèmes experts commercialisés par la Digital Equipment Corporation en 1982, puis une **science à part entière** dès le début des années 1990, où elle intégra des pans entiers d'autres domaines (statistiques et probabilités, théorie du contrôle, logique formelle...), l'IA a désormais gagné ses lettres de noblesse.

Aujourd'hui, l'IA est à l'honneur dans des secteurs aussi divers que le **génie industriel** (planification de processus, logistique, allocation optimale de ressources), la **médecine** (systèmes experts diagnostiquant ou aidant à diagnostiquer des maladies ou syndromes, assistance à la chirurgie [7]), les **banques et assurances** (estimation de risques, *trading*, et bien d'autres), l'**industrie** (conception et vérification de processeurs, logiciels embarqués, *data mining* et *business intelligence*, analyse de l'actualité et veille technologique, etc.) ... et la liste est loin d'être exhaustive. Il n'est donc pas absurde, même si le sujet peut *a priori* sembler moins sérieux, de se demander quel est le statut de l'intelligence artificielle dans un des domaines où le terme est constamment employé - à savoir celui des jeux vidéo.

1.3 Problème, formulation et objectif

Dans le cadre de notre projet Innovation, nous avons choisi de nous atteler au problème suivant : *tenter d'intégrer de l'intelligence artificielle dans un jeu vidéo est-il viable ? Si oui, comment s'y prendre ? Et si non, pour quelles raisons ?*

Pour répondre à ces questions, il nous a fallu dans un premier temps - et c'est l'objet de la première partie de ce rapport - dresser un état de l'art des utilisations de l'IA dans le domaine vidéoludique ; après quoi nous nous sommes penchés sur les principes de différents types d'IA existants, dans l'optique de déterminer s'ils étaient applicables à notre projet - le résultat de cette recherche est présenté dans la seconde partie.
Une fois ceci mené à bien, nous avons pu décider en relative connaissance de cause de l'orientation effective de notre projet, à savoir *ce que nous allions faire, et de quelle manière* ; les choix effectués et les raisons les ayant motivés forment la troisième et dernière partie de ce document.

7. Voir par exemple HipNav, développé à l'Université de Carnegie Mellon (`http://www.ri.cmu.edu/publication_view.html?pub_id=1773`)

Chapitre 2

Quelle intelligence artificielle et algorithmes sont actuellement utilisés dans les jeux vidéos ?

2.1 *Scripting* vs. IA

> « L'IA commence là où l'informatique classique s'arrête : tout problème pour lequel il n'existe pas d'algorithme connu ou raisonnable permettant de le résoudre relève a priori de l'IA. »
>
> (Jean-Louis Laurière [1])

> « Les principales composantes d'un système d'IA doivent être les connaissances, le raisonnement, la compréhension du langage naturel et l'apprentissage. »
>
> (Alan Turing [2])

Par *scripting*, on désigne généralement un code ou programme qui définit le comportement du personnage d'une manière prédéterminée par le programmeur, comme dans un roman. Au mieux, son « rôle » (dans l'histoire) peut varier selon différents paramètres : il s'agit, en quelque sorte, d'une histoire proposant plusieurs fins.

- Un exemple de script pur : un coéquipier va dès le début de la partie aller vers la porte ; lorsque le joueur arrive à côté de lui, un ennemi enfonce la porte et écrase ledit coéquipier.
- Un exemple d'IA scriptée : si l'on tue un policier, les autres policiers vont venir en représailles ; sinon, ils ne feront qu'ignorer le joueur. Ou bien, “quelle va être la réaction des *bots* si une grenade tombe à leurs pieds ?”

---

1. Ancien professeur d'informatique de l'Université Pierre et Marie Curie (Paris VI), il a été l'un des pionniers de la recherche en intelligence artificielle (IA), notamment dans les domaines de la résolution de problèmes et de la représentation des connaissances.
2. Mathématicien et logicien britannique de la première moitié du XXe siècle, Turing fut l'un des pères fondateurs de l'informatique théorique, en particulier dans le domaine de la calculabilité, avec la *machine de Turing*, modèle abstrait ayant inspiré les premiers ordinateurs.

D'une certaine manière, on peut considérer que le scripting est *combinatoire* : à un état donné correspondra une réaction.

Le principe de l'intelligence artificielle appliquée aux jeux vidéo, en revanche, est d'obtenir un personnage qui s'adapte *totalement* au comportement du joueur. Il n'y a pas un nombre fini de scénarios possibles, compte tenu de déclencheurs, comme ce serait le cas dans une IA scriptée, mais un ajustement de son attitude. Si l'on peut envisager une IA combinatoire, il est également possible d'atteindre une IA *séquentielle* : le comportement du personnage dans un état donné dépend de ce qui s'est déroulé auparavant.

Un autre fondamental de l'intelligence artificielle que nous recherchons est le concept d'**apprentissage** : dans une IA scriptée, le personnage ne fait que ce qu'on lui a dit de faire, ne peut en aucune manière innover et effectuer une action complètement imprévue par son concepteur. Une « véritable » IA, elle, peut s'adapter à son environnement, l'influence du programmeur n'étant plus qu'un biais dans sa manière de s'adapter - l'équivalent d'une « configuration d'esprit initiale ».

Exemple de jeu où il y a une réelle IA : il semblerait que dans *Super Smash Bros.^{TM}Melee*, l'ordinateur puisse adapter son mode de jeu à celui du joueur (IA évolutive).

2.2 Exploration d'arbres et algorithmes de recherche

Les concepts introduits dans cette section, communs à beaucoup domaines de l'informatique, sont nécessaires à la bonne compréhension des parties suivantes. En particulier, il sera fait référence ultérieurement (dans la partie 2.3 notamment) aux notions de parcours et de recherche dans un arbre, ainsi qu'aux différents algorithmes présentés.

2.2.1 Généralités

Parcourir un arbre consiste à considérer successivement l'ensemble de ses nœuds, dans un ordre déterminé. Il s'agit par conséquent d'une fonction qui prend en argument un arbre et qui renvoie une liste de nœuds (même si celle-ci n'est souvent pas construite explicitement durant le parcours).
On distingue principalement deux classes de parcours : le *parcours en largeur* et le *parcours en profondeur*.

Parcours en largeur (*Breadth-First Search*, ou BFS)

Le parcours en largeur consiste à parcourir l'arbre niveau par niveau. Dans chaque niveau, les nœuds sont parcourus de la gauche vers la droite. Le parcours en largeur de l'arbre de la Figure 2.1 considère ainsi les nœuds dans l'ordre

$$[0, 1, 8, 2, 4, 9, 13, 3, 5, 6, 10, 14, 15, 7, 11, 12]$$

Parcours en profondeur (*Depth-First Search*, ou DFS)

Les parcours en profondeur se définissent souvent récursivement. Le parcours consiste à traiter la racine de l'arbre et à parcourir récursivement les sous-arbres

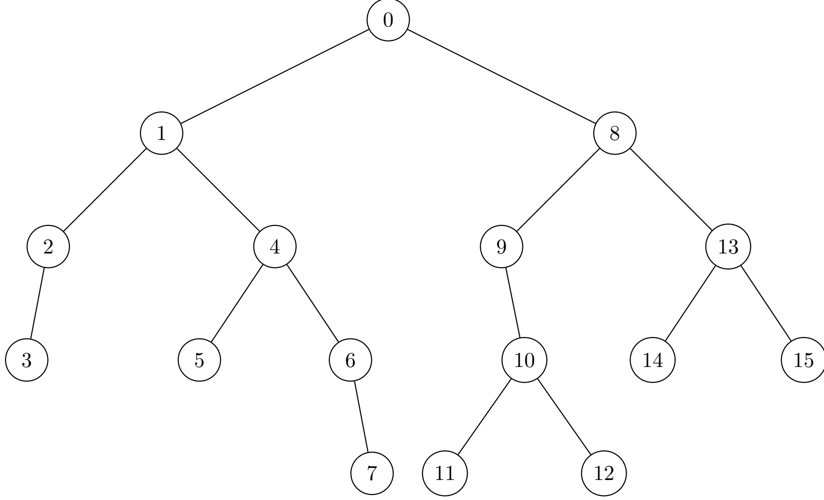

Figure 2.1 – Arbre à parcourir

gauche et droit de la racine. Il existe 3 types de parcours (préfixe, infixe et suffixe) et ils se distinguent par l'ordre dans lequel sont faits ces traitements :

Dans le *parcours préfixe*, la racine est traitée avant les appels récursifs sur les sous-arbres gauche et droit (effectués dans cet ordre). Le parcours préfixe de l'arbre précédent parcourt ainsi les nœuds dans l'ordre

$$[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]$$

Dans le *parcours infixe* (spécifique aux arbres binaires, i.e où chaque nœud non terminal a exactement deux fils), le traitement de la racine est effectué entre les appels sur les sous-arbres gauche et droit. Le parcours infixe de l'arbre ci-dessus parcourt les nœuds dans l'ordre

$$[3, 2, 1, 5, 4, 6, 7, 0, 9, 11, 10, 12, 8, 14, 13, 15]$$

Dans le *parcours suffixe*, la racine est traitée après les appels récursifs sur les sous-arbres gauche et droit (dans cet ordre). Le parcours suffixe de l'arbre ci-dessus parcourt alors les nœuds dans l'ordre

$$[3, 2, 5, 7, 6, 4, 1, 11, 12, 10, 9, 14, 15, 13, 8, 0]$$

2.2.2 Heuristiques, A*

L'algorithme A^* est une forme d'*exploration meilleur d'abord*[3]. Le principe d'une telle exploration est le suivant : Considérons un arbre T dont les nœuds n sont des états. On cherche à se rendre à un *état objectif*, les branches de l'arbre étant autant de façons de se rendre de l'état courant à l'*état objectif*. Bien sûr, il ne suffit pas de s'y rendre par n'importe quel chemin, mais par celui

3. BFS, ou *Best-First Search*.

qui coûte le moins : en effet, on affecte à chaque branche de l'arbre un *coût* qui dépend des paramètres du problème. Pour savoir quel nœud développer dans le parcours de l'arbre, on se donne une *fonction d'évaluation* $f(n)$. Habituellement, celle-ci mesure la distance au but et sélectionne donc le nœud caractérisé par l'évaluation la plus faible. Le problème de l'*exploration meilleur d'abord* réside dans le fait qu'elle ne peut sélectionner que le nœud qui semble le meilleur compte tenu de la *fonction d'évaluation*, mais il n'y a aucune raison pour que ce nœud appartienne effectivement au chemin optimal. La *fonction heuristique* $h(n)$ va jouer un rôle primordial dans ce type d'algorithme. Par définition :

$$h(n) = \text{ coût estimé du chemin le moins coûteux menant du nœud n au nœud objectif.}$$

L'algorithme A^* va combiner l'utilisation de la fonction heuristique $h(n)$ et de la fonction $g(n)$ définie comme suit :

$$g(n) = \text{ coût estimé du chemin pour se rendre au nœud } n$$

La fonction d'évaluation vaut ainsi :

$$f(n) = g(n) + h(n)$$

De sorte que :

$$f(n) = \text{ coût estimé de la solution la moins onéreuse passant par } n$$

Le principe de l'algorithme A^* (voir Listing 1 pour une implémentation en pseudocode) est donc simple : pour atteindre le nœud objectif avec un coût minimal, il semble raisonnable de développer le nœud n qui minimise $f(n)$. On peut d'ailleurs prouver, avec certaines conditions sur $h(n)$, que l'exploration A^* est optimale (il suffit en fait de choisir $h(n)$ ne surestimant jamais le coût pour atteindre le nœud objectif).

2.2.3 Min-Max, Alpha-Beta

Ces algorithmes [Tor08] servent à l'exploration d'un graphe représentant un jeu opposant deux joueurs. Ceux-ci, J_1 et J_2, jouent à tour de rôle. On se place du point de vue de J_1 pour définir la victoire et la défaite, et on crée un arbre correspondant à la situation où c'est à J_1 de jouer :

- La racine est l'état actuel du jeu
- Les nœuds de profondeur paire sont les situations où c'est à J_1 de jouer
- Les nœuds de profondeur impaire sont les situations où c'est à J_2 de jouer
- Les arcs sont des coups possibles, et relient chacun une situation où le coup peut être joué à la situation résultant de ce coup
- Les feuilles sont les états de fin de partie, pouvant être (toujours selon le point de vue de J_1) gagnant, perdant ou nul
- Une partie possible du jeu est un chemin de la racine à une des feuilles

Algorithme Min-Max

D'après le point de vue que l'on a pris - celui de J_1 - on attribue une valeur à chaque feuille de l'arbre :

- Les états gagnants valent $+1$

Listing 1 Algorithme A*

Input :

- n_i {état initial}
- n_f {état final}
- N_d {liste des nœuds développés}
- N_c {liste des nœuds *potentiels* du chemin optimal}
- f {fonction d'évaluation}

Statique :

- $n \leftarrow n_i$
- $N_d \leftarrow [n_i]_i$
- $N_c \leftarrow [n_i]_i$

while $n \neq n_f$ **do**
  $n \leftarrow \text{argmin}_{n' \in N_d} f(n')$ {Choix du nœud le plus "proche" du nœud final}
  $N_d \leftarrow N_d \cup \texttt{Sons}(n) \setminus \{n\}$
  $N_c \leftarrow N_c \cup \{n\}$
end while
$\Gamma \leftarrow n_f$ {On repart du nœud final}
while $n_i \notin \Gamma$ **do**
  $\Gamma \leftarrow \texttt{Father}(\texttt{Head}(\Gamma)) \cup \Gamma$ {Pour reconstruire le chemin "à l'envers"}
end while

- Les états perdants valent -1
- Les états nuls valent 0

Puis on fait « remonter » ces valeurs de nœud en nœud, pour à la fin pouvoir dire quelle action le joueur J_1 doit effectuer s'il veut gagner. Pour ce faire :

- Aux nœuds de profondeur paire, c'est à J_1 de jouer ; il choisit le coup amenant à l'état de plus grande valeur. On attribue donc à ces nœuds le Max des valeurs des nœuds fils.
- Aux nœuds de profondeur impaire, c'est J_2 qui joue et choisit le coup amenant à l'état de plus petite valeur. On attribue donc à ces nœuds le Min des valeurs des nœuds fils.

Le joueur J_1 devra ensuite choisir les coups l'amenant à des états de plus grande valeur possible.

Exemple : Le jeu des allumettes. Ce célèbre jeu consiste à faire retirer des allumettes d'un tas par deux joueurs, chacun à leur tour. A chaque tour, le joueur peut retirer 1, 2 ou 3 allumettes. Celui qui retire la dernière allumette a perdu.

L'arbre de la Figure 2.2 décrit le jeu avec une quantité initiale de 3 allumettes. Les chiffres aux nœuds correspondent au nombre d'allumettes restantes, et ceux aux arcs au nombre d'allumettes retirées par coup.

On attribue ensuite à chaque feuille une note (+1 ou -1) en fonction du résultat de la partie (Figure 2.3).

Puis par la méthode décrite plus haut on attribue une note aux nœuds, en « remontant ». En vert, les notes sont données en prenant le minimum des notes des fils (coup de J_2) ; en bleu, en prenant le Max (coup de J_1) (Figure 2.4).

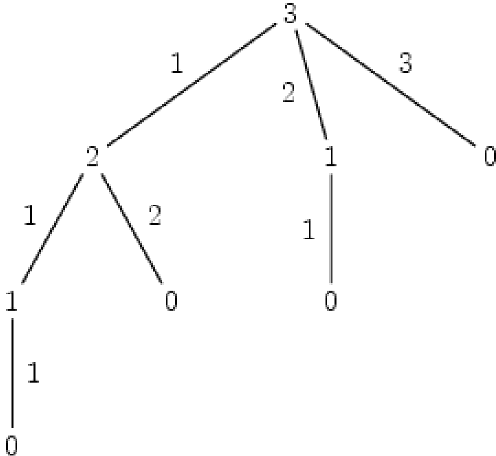

FIGURE 2.2 – Jeu des allumettes

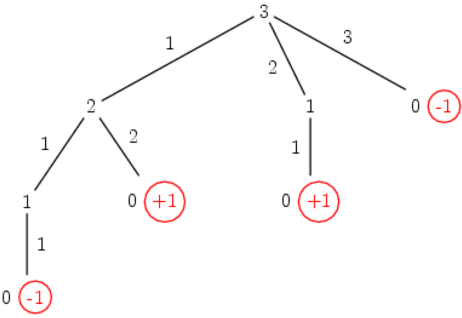

FIGURE 2.3 – Attribution des notes finales

Algorithme Alpha-Bêta

Le problème de la méthode *Min-Max* est qu'elle nécessite de développer et de parcourir entièrement l'arbre pour avoir un résultat. Cela demande un long temps de calcul. Mais il existe des méthodes pour « élaguer » l'arbre, et donc réduire le nombre de calculs à effectuer : en particulier, les *coupes alpha* et *bêta*.

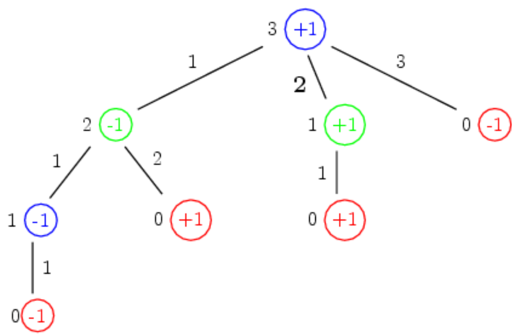

FIGURE 2.4 – “Remontée” des notes

Coupe Alpha

Plaçons-nous sur un nœud où c'est à J_1 de jouer : il choisit le nœud fils ayant la note maximale. Supposons que nous connaissions les notes a_i des n fils de ce nœud, exceptée celle du dernier, a_n. Pour ce fils n, qui est un nœud de J_2, nous connaissons la note b_1 de son premier fils, mais pas les autres notes b_j. On sait donc que la note a_n sera inférieure ou égale à b_1, puisque J_2 cherche à minimiser le score de l'état. Il suffit alors que l'un des a_i soit supérieur b_1 pour que l'on déduise que le nœud de J_1 ne choisira pas le coup n, qui donnerait un état dont la note a_n ne dépassera pas b_1. On peut alors se passer de calculer les b_j restants, et donc tous les sous-arbres correspondants.

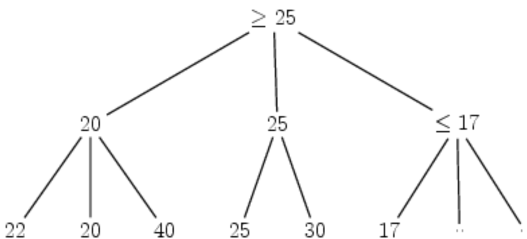

FIGURE 2.5 – Exemple de coupe alpha

Exemple : Le troisième fils (Figure 2.5) ayant une note forcément inférieure à 17 (c'est J_2 qui choisit et donc qui veut minimiser le score), J_1 ne le choisira pas puisqu'il peut choisir 20 ou 25. Inutile donc de développer le sous arbre de ce troisième fils.

Coupe Bêta

Le même principe est appliqué à un nœud où c'est à J_2 de jouer. Les règles de supériorité et d'infériorité sont inversées par rapport à la coupe alpha (voir

Figure 2.6, mais le résultat est le même : l'élagage d'un sous-arbre.

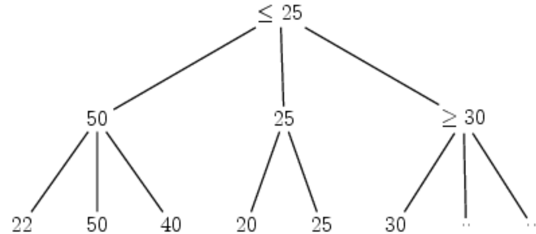

FIGURE 2.6 – Exemple de coupe bêta

2.2.4 UCB et UCT

Lorsque le facteur de branchement devient trop important (comme c'est le cas dans le jeu de go), il devient nécessaire d'avoir une stratégie de choix des actions à envisager qui favorise des parties de l'arbre, que l'on espère les plus prometteuses, aux dépens des parties jugées moins intéressantes (le fameux « compromis exploration vs. exploitation » énoncé dans la section 4.4).

L'algorithme *UCB* (Upper Confidence Bounds) (voir Listing 2), introduit en 2002, résout ce compromis en choisissant à chaque tour, parmi toutes les actions possibles, celle qui maximise la somme d'un terme d'exploitation (moyenne des gains obtenus en choisissant cette action) et d'un terme d'exploration (terme de variance d'autant plus élevée que cette action a été choisie peu souvent dans le passé) ([AC02], Chapitre 16).

On obtient ainsi un compromis tel que, lorsque l'on joue un grand nombre de fois, la probabilité de choisir un autre bras que le bras optimal tend vers zéro avec une vitesse de convergence optimale (en $\frac{\ln n}{n}$, où n est le nombre de parties jouées). Contrairement à l'algorithme alpha-bêta (ou à min-max, dont il découle), UCB choisit l'action qui *pourrait se révéler la meilleure*[4]. On parle de *stratégie optimiste dans l'incertain*.

Plus précisément, ces algorithmes dérivent de recherches sur le problème du bandit multi-bras (extension du problème du bandit à deux bras, voir section 4.4). Un problème de bandit à K bras est défini par des variables aléatoires $X_{i,n}$, avec $i \leq K$ et $1 \leq n$ (où i est le numéro d'un bras, et n l'instant considéré). Les tirages successifs du bras i rapportent des renforcements $X_{i,1}, X_{i,2}, \ldots$ indépendants et identiquement distribués (iid) selon une loi de moyenne μ_i.

Une *stratégie d'allocation* est un algorithme déterminant le bras à jouer à l'instant $n+1$ en se basant sur la séquence des bras déjà tirés et des renforcements observés. Soit $T_i(n)$ le nombre de fois que le bras i a été tiré durant les n premiers

4. Selon la moyenne augmentée de la largeur de l'intervalle de confiance.

Listing 2 UCB

Input :
- $[< x_j >]_j$ {Renforcements moyens obtenus pour chaque bras}
- $[T_j(n)]_j$ {Nombre de tirages de chaque bras}
- N {Nombre de tirages déjà effectués}

for all j **do**
  JOUER($\texttt{Arm}_j$) {Initialisation}
end for
repeat
  $j \leftarrow \text{argmax}_j \left(< x_j > + \sqrt{2\frac{\ln n}{T_j(n)}}\right)$
  JOUER($\texttt{Arm}_j$)
until Fin du jeu

instants. Le *regret* associé à la stratégie est alors défini par :

$$\text{Regret}(n) = \mu^* n - \sum_{1 \leq j \leq K} \mu_j \text{E}\left[T_j(n)\right]$$

où $\mu^* = \max_{1 \leq i \leq K} \mu_i$.

Le regret est ainsi l'espérance de la perte due au fait que la stratégie d'allocation ne joue pas systématiquement le meilleur bras. UCB est une stratégie d'allocation selon laquelle le bras optimal est joué exponentiellement plus souvent que les autres. Le regret obtenu à l'aide de cette stratégie est alors le meilleur possible, avec $\text{Regret}(n) = O(\ln(n))$ [5]. Cette stratégie associe une valeur appelée *indice de confiance supérieur* à chaque bras. Des travaux ultérieurs ont simplifié le calcul de cet indice, en se basant sur l'histoire passée des tirages.

L'algorithme UCB ne peut cependant pas être utilisé tel quel dans notre cas, car les gains peuvent subvenir bien après l'action choisie. L'algorithme UCT *(Upper Confidence bounds in Trees)* utilisee UCB de façon récursive pour estimer la valeur des nœuds et de la racine dans l'arbre des coups possibles. À chaque nœud exploré de l'arbre est associé un algorithme de type UCB sélectionnant la prochaine action à entreprendre. Le choix d'une séquence d'actions est ainsi décomposé en un ensemble de problèmes plus simples selon une méthode de type Monte-Carlo dans laquelle les branches les plus prometteuses sont explorées en priorité.

Dans ses grandes lignes, l'algorithme UCT développe et explore l'arbre de manière asymétrique en réalisant un compromis entre l'exploration des branches qui semblent les meilleures, compte tenu des feuilles déjà explorées, et l'exploration des branches encore mal connues. Des épisodes sont lancés depuis la racine jusqu'à des feuilles. En chaque nœud rencontré, s'il reste des successeurs non explorés, l'algorithme en choisit un aléatoirement. Si tous les nœuds fils ont déjà été visités au moins une fois, l'algorithme choisit le nœud n maximisant la valeur UCB de j. Quand une feuille est atteinte, sa valeur est intégrée dans la valeur moyenne des nœuds traversés.

5. Résultat démontré par Lai et Robbins en 1985 [LR85]. Voir aussi `http://researchers.lille.inria.fr/~munos/master-mva/lecture03.pdf`

La fonction `EXPLORER` sélectionne et effectue des actions récursivement jusqu'à ce qu'une condition de terminaison soit satisfaite : soit que l'état atteint soit une feuille, soit que la profondeur limite ait été atteinte.

La fonction `MISE-A-JOUR` utilise le gain total $Q_{UCT}(s, a)$ pour ajuster la valeur estimée pour la paire *(état,action)* à la profondeur donnée, et le nombre de visites de la paire est incrémenté. Plus formellement, si on note $s_1, a_1, s_2, a_2, \ldots s_T$ la séquence d'états et d'actions correspondant à un épisode exploré par UCT à partir de s_1, alors `MISE-À-JOUR` met à jour chaque paire *(état,action)* selon :

$$n(s_t, a_t) = n(s_t, a_t) + 1$$
$$Q_{UCT}(s_t, a_t) = Q_{UCT}(s_t, a_t) + \left(R_T - \frac{Q_{UCT}(s_t, a_t)}{n(s_t, a_t)}\right)$$

où R_T est le gain cumulé mesuré depuis (s_t, a_t). On initialise les paires *(s,a)* n'ayant encore jamais été rencontrées par $n(s_t, a_t) = 1$ et $Q_{UCT}(s_t, a_t) = R_t$.

La fonction `SELECT-ACTION(etat, profondeur)` détermine les actions à essayer, de la même manière qu'UCB. Chaque choix d'action, pour chaque nœud s exploré, est traité comme un problème de bandit multi-bras indépendant. Un bras correspond à une action a possible, et le gain associé au gain cumulé mesuré (jusque-là) des chemins explorés partant du nœud est $Q_{UCT}(s_t, a_t, d(s))$ (où $d(s)$ est la profondeur du nœud s). Le bras (action) associé à la plus grande valeur de

$$Q_{UCT}\left(s_t, a_t, d(s)\right) + C\sqrt{\frac{\ln n(s_t)}{n(s_t, a_t)}} \tag{2.1}$$

est sélectionné, où $C > 0$ est une constante "appropriée" (la *Constante UCT*), $n(s_t)$ le nombre de visites du nœud s jusqu'à l'instant t, et $n(s_t, a_t)$ le nombre de fois où l'action a_t a été sélectionnée dans l'état s_t, jusqu'à ce même instant.

On dispose ainsi de toutes les fonctions intervenant dans la définition d'`EXPLORER` :

On peut alors définir l'algorithme UCT [Gel07], [Saf08] :

1. À partir d'un nœud racine, parcourir l'arbre jusqu'à atteindre une feuille vérifiant le critère (2.1)
2. Sélectionner une action et développer le nœud fils de la feuille
3. Exécuter une simulation aléatoire à partir de ce nœud fils jusqu'à la fin du jeu et obtenir le retour, à savoir le résultat du jeu
4. Mettre à jour la valeur moyenne des nœuds parcourus avec le retour obtenu.

La valeur du nœud racine considéré converge alors vers la valeur *min-max* de l'arbre de jeu.

2.3 Dans les jeux « classiques » (échecs, go . . .)

Apprentissage par renforcement Il est à noter que si le domaine du jeu pose *a priori* un problème d'apprentissage, étant donné que l'influence de l'adversaire sur l'environnement rend celui-ci instationnaire (ou dynamique), de

Listing 3 EXPLORER(état, profondeur) : renvoie la valeur de l'état

```
Input :
  – s {état}
  – d_s {profondeur de l'état}
Output : q {valeur de l'état}
  if TERMINAL(s) then
    return 0
  else
    if FEUILLE(s) then
      return ÉVALUE(s)
    end if
  end if
  a = SELECT − ACTION(s, d_s)
  (s', r) = ACTION − SIMULEE(s, a) {(état suivant, renforcement)}
  q = r + γ · EXPLORER(s', d_s + 1)
  MISE − A − JOUR(s, a, q, d_s)
  return q
```

nombreuses tentatives de systèmes d'apprentissage par renforcement ont eu lieu, dans un grand nombre de jeux. Les jeux présentent en effet l'avantage de disposer de règles clairement définies et aisément assimilables par un programme, et de pouvoir faire l'objet d'une modélisation abstraite relativement simple, dans laquelle intervient un nombre assez faible de facteurs.

Échecs

Le jeu d'échecs comporte plusieurs caractéristiques qui en font un sujet intéressant dans le cadre de l'intelligence artificielle. Plus précisément :

- il est défini par un ensemble de règles strictes et non ouvertes à l'interprétation
- des matches peuvent être organisés entre ordinateurs et humain pour mesurer la performance.
- beaucoup de gens savent y jouer, et la littérature lui étant consacrée (stratégies, analyse et exemples de parties) abonde
- la complexité du jeu est assez grande pour empêcher une analyse exhaustive de tout le jeu (pas de *brute force* possible)

Une date est à retenir dans l'histoire des échecs : en 1997, Deep Blue (IBM) bat le champion international Garry Kasparov. Depuis, toutes les confrontations tournent systématiquement à l'avantage de la machine.

En raison du nombre de choix possible à chaque tour dans un jeu d'échec, une méthode « gloutonne » consistant à parcourir l'arbre *(noeuds = états, arêtes = actions)*, comme le fait par exemple le célèbre algorithme *min-max*[6] , est, paradoxalement, vouée à l'échec.

Deux méthodes sont envisageables pour dépasser ce problème :

- limiter la profondeur et utiliser une heuristique

6. Les algorithmes *alpha-bêta* et *min-max* seront développés plus avant dans la section 2.2.3.

– élaguer certaines branches de l'arbre (sans heuristique : on coupe « sans risque ») en utilisant l'algorithme *alpha-bêta* [6].

En pratique, c'est la seconde solution qui est privilégiée, et le facteur de branchement (nombre d'actions possibles en chaque état) vaut ici 40.

Puissance 4

Ce jeu est résolu. Cela signifie que l'on a montré qu'il existait toujours une stratégie gagnante pour le joueur qui commence à jouer, représentable par exemple sous la forme d'un arbre de décision spécifiant le coup à jouer à chaque étape, quel que soit celui choisi par l'adversaire. Cette stratégie peut, jusqu'à une certaine taille de grille (typiquement, une grille de 6×7 cases) être intégralement calculée et stockée en mémoire [7] - un joueur artificiel n'ayant plus alors qu'à se conformer à cette stratégie pour gagner systématiquement.

Jeu de dames

Le jeu de dames américain est une variante du jeu de dames français (ou international), dont il diffère par quelques points, en particulier la taille du plateau ($8 \times$ *au lieu de* 10×10*) et l'impossibilité de "manger" en arrière.*

La plus ancienne application réussie est celle du jeu de dames américain due à Samuel en 1959 [Sam59]. Le programme apprenait une fonction d'évaluation $V(s)$ représentée par une fonction linéaire d'un certain nombre de facteurs déterminés par Samuel. Il employait un mécanisme d'apprentissage similaire à celui de l'algorithme d'*itération de valeur*, des *différences temporelles* et du *Q-learning* (cf. section 3.4).

En ce qui concerne le jeu de dames international, le programme Chinook basé sur l'algorithme alpha-bêta [6] est devenu champion du monde en 1994. Celui-ci dispose en mémoire d'une bibliothèque de fins de partie (tous les damiers comportant 8 pièces ou moins), qu'il utilise afin de déterminer le meilleur mouvement à sa disposition (dans le but de se ramener à l'une des configurations finales connues).

Backgammon

Un autre succès, plus récent, est celui de G.Tesauro [8]dans le domaine du backgammon [Tes95], un jeu de table à deux joueurs dont le but est d'être le premier à amener tous ses pions à la sortie du plateau. Ce jeu comporte environ 10^{20} états, ce qui rend impossible une méthode fondée sur l'utilisation d'une table d'états. Il est donc indispensable d'utiliser une méthode de généralisation dans l'espace des états. Tesauro a employé un perceptron multicouche (cf. 3.2) à une couche cachée avec apprentissage par rétropropagation de gradient, pour réaliser un système d'estimation de la fonction de valeur :

Position sur le jeu → probabilité de victoire pour le joueur courant

7. cf. `http://www.enseignement.polytechnique.fr/informatique/profs/Fabrice.Le-Fessant/PI09/Francois.Pottier.pdf`

8. Gerald Tesauro, chercheur à IBM, s'intéresse notamment à l'apprentissage par renforcement dans le système nerveux et aux applications des réseaux neuronaux en finance et dans la détection des virus informatiques.

Une première version de base de l'algorithme appelé TD-Gammon ne comportait aucune connaissance spécifique du domaine, tandis que les versions ultérieures utilisaient des connaissances propres à certaines positions de jeu. Pour toutes ces versions, l'apprentissage fut réalisé par simulation de jeu de l'ordinateur contre lui-même. Remarquablement, aucune stratégie d'exploration n'était utilisée, bien que l'algorithme choisisse invariablement le coup apparemment le meilleur. Cela ne pose pas de problème au backgammon car les situations de jeu obtenues dépendent en partie d'un tirage aux dés - cette composante aléatoire suffisant théoriquement à garantir qu'après un temps suffisamment long, tous les états seront visités. De plus, il s'agit d'un jeu dans lequel les parties terminent en un temps borné (les pions ne pouvant qu'avancer, d'au moins une case à chaque tour), ce qui assure que des renforcements sont reçus assez fréquemment. TD-Gammon se place parmi les meilleurs joueurs mondiaux.

Jeu de go

Le go est un jeu à deux joueurs d'origine chinoise. Deux adversaires placent à tour de rôle des pions blancs ou noirs sur les intersections d'un tableau appelé *go-ban* de dimension normale 19×19, mais qui se joue aussi sur des plateaux de dimensions plus réduites (9×9 ou 13×13). Il s'agit d'un jeu à *information complète* - autrement dit, il ne comporte aucun hasard, et chaque joueur connaît à chaque instant toutes les possibilités de jeu. Le vainqueur est le joueur ayant réussi à délimiter par ses pions un territoire plus vaste que celui de son adversaire.

Ce jeu est l'un des derniers grands jeux (avec, entre autres, le poker) dans lesquels les experts humains restent très supérieurs à la machine. En ce qui concerne le go, deux raisons au moins peuvent être invoquées : d'une part, on ne dispose pas (jusqu'en 2008 tout du moins, date de la source [AC02]) de fonction d'évaluation d'un go-ban satisfaisante, qui reflète assez fidèlement la force d'une position. D'autre part, le facteur de branchement du jeu, de l'ordre de 200 en milieu de partie, excède le facteur de branchement de jeux comme les échecs (de l'ordre de 30-40 environ) et ne se prête pas facilement à des approches traditionnelles comme l'algorithme alpha-beta qui suppose une exploration complète (même si des élagages sont recherchés) de l'arbre de jeu jusqu'à une profondeur seuil.

À ces deux obstacles, l'approche *UCT* (cf. 2.2.4) permet d'apporter une réponse. La première idée est de remplacer l'énigmatique fonction d'évaluation de position, si difficile à définir, par une exploration par *échantillonnage de Monte-Carlo* de parties possibles jusqu'à leur conclusion : chaque partie jouée permet d'obtenir un résultat sûr (perte ou gain), et c'est la moyenne de gain de ces parties qui joue alors le rôle de fonction d'évaluation. Par ailleurs, et c'est la seconde idée, au lieu d'une exploration complètement aléatoire, l'approche UCT permet de concentrer les explorations vers les *régions les plus prometteuses de l'arbre de jeu*. Finalement, un avantage de cette technique est d'être intrinsèquement *anytime*, c'est-à-dire de pouvoir fournir une réponse à tout moment, la précision de cette réponse augmentant au cours du temps.

Les systèmes de jeu de go les plus performants, à l'heure actuelle, utilisent la technique UCT. C'est le cas par exemple de MoGo [9], qui a remporté le 26

9. Site officiel de MoGo : `http://www.lri.fr/~teytaud/mogo.html`

mars 2008 la première victoire homologuée, non-blitz [10], opposant une machine à un maître du go.

2.4 Dans les jeux de stratégie en temps réel

Dans les jeux video de type RTS (*Real-Time Strategy*, ou *jeu de stratégie en temps réel*), l'IA se divise en plusieurs modules, ou « couches ». L'un des modules les plus fondamentaux est ainsi le système de *pathfinding* [11] : contrairement à ce qui se produit dans d'autres types de jeux, il est en effet nécessaire dans les RTS de trouver en permanence une solution de mouvement pour des centaines d'unités sur une carte, et ceci en une fraction de seconde (faute de quoi le caractère "temps réel" du jeu serait compromis). En outre, il ne s'agit plus seulement ici de trouver un chemin d'un point A à un point B, mais également de détecter les collisions potentielles et de les éviter (dans la mesure du possible). Les algorithmes permettant ceci sont typiquement fondés sur une représentation de la carte sous forme de grille rectangulaire de maillage uniforme (voir Figure 2.7), cette subdivision étant alors équivalente à un graphe.

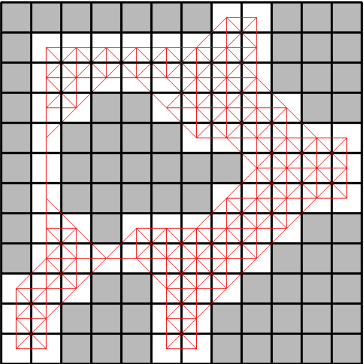

FIGURE 2.7 – Représentation de l'univers du jeu (carte) dans un RTS

Sur la Figure 2.7, les cases blanches et grises représentent respectivement les zones accessible et inaccessible, les traits rouges les déplacements possibles.

À chaque étape, l'IA calcule le nombre de « cases » nécessaires pour le déplacement des troupes et, dans le cas où le chemin serait trop étroit, donne l'ordre à certaines unités (des priorités sont définies au préalable selon l'importance des unités et le type de stratégie décidée) de s'arrêter afin de ne pas ralentir le déplacement collectif.

À des niveaux plus élevés de l'IA, on peut trouver des modules qui concernent des aspects globaux du jeu (tels que l'économie, développement ...) [12] mais surtout un module en charge d'analyser la carte et de déterminer une stratégie adaptée. C'est ce dernier qui décide, en fonction des propriétés du terrain, d'une

10. C'est-à-dire sans que le temps de réflexion à chaque coup ne soit limité.
11. Voir à ce sujet la section 2.2, qui présente l'algorithme A* *(A star)*.
12. e.g, dans des jeux tels que ceux de la série des *Age of Empires*

configuration de jeu - ainsi, si le campement (position de départ) est situé sur une île, la possession d'une flotte puissante pourrait être établie comme objectif prioritaire.

2.5 Dans les jeux de tir à la première personne

Dans les jeux vidéo de type FPS (*First-Person Shooter*, ou *jeu de tir subjectif*), l'IA généralement mise en œuvre est également structurée en couches superposées : les couches de bas niveau gèrent les tâches les plus simples, comme déterminer le chemin optimal, tandis que les niveaux supérieurs sont responsables du raisonnement tactique - en sélectionnant par exemple les actions d'un agent selon la stratégie qu'il a choisi d'adopter (agressive, défensive...) et ses caractéristiques.

Les systèmes de *pathfinding* reposent la plupart du temps sur des graphes décrivant l'univers du jeu, où chaque sommet représente un emplacement logique (une pièce dans un bâtiment, par exemple). Pour se déplacer jusqu'à un point donné de la carte, l'agent reçoit la succession de sommets par lesquels il doit passer pour atteindre son objectif. En se déplaçant entre deux points de navigation (sommets du graphe), ce qui correspond à un déplacement logique, il peut aussi être nécessaire de calculer des chemins locaux, c'est-à-dire les déplacements "physiques" précis entre deux sommets, et aussi d'éviter les obstacles dynamiques qui pourraient apparaître. Le système d'animation effectue une certaine séquence de mouvement à la vitesse choisie. Il est aussi capable de gérer plusieurs séquences à partir du moments où elles mobilisent différentes parties du « corps ». Par exemple, un soldat peut courir et viser l'ennemi puis tirer et recharger tout en continuant à courir. Les FPS utilisent souvent le système cinématique inversé (*Inverted Kinematics System*). Grâce à ce dernier, l'animation permettant par exemple le mouvement du bras pour récupérer un objet sur une table peut être pré-calculée. Ici encore , c'est la « couche supérieure » qui décide du comportement de l'agent (par exemple, est-il préférable de patrouiller dans une zone donnée, ou plutôt de parcourir toute la carte à la recherche d'ennemis ?).

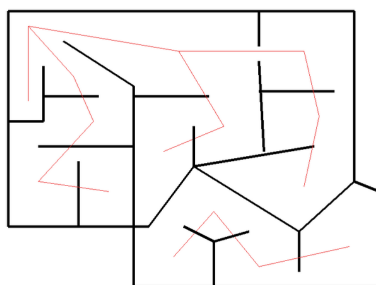

FIGURE 2.8 – Représentation de l'univers du jeu (carte) dans un FPS

Ici (Figure 2.8), chaque sommet du graphe constitue une emplacement logique sur la carte (typiquement le centre d'une pièce). L'existence d'une arête entre deux sommets signifie qu'il y a au moins un chemin les reliant . Le déplacement

d'un sommet à un autre se fait en utilisant des chemins locaux qui prennent en compte les obstacles statiques (murs) et dynamiques (autres agents)

Une fois que le système d'IA a choisi le comportement le plus approprié pour une situation donnée, un module inférieur doit déterminer la meilleure tactique pour accomplir la tâche demandée. Par exemple, si l'agent reçoit l'ordre de se battre, il tente de déterminer la meilleure approche actuelle : vaut-il mieux se cacher et attendre que l'adversaire se découvre, ou attaquer ce dernier de manière aléatoire (dans le cas où l'agent a l'information que son adversaire n'a plus beaucoup de points de vie par exemple). Globalement, c'est selon ce principe que fonctionnent la plupart des IA des FPS modernes (avec des degrés de sophistication différents). Néanmoins, il est intéressant de noter quelques utilisations de méthodes d'apprentissage par renforcement dans le jeu « Counter-Strike » [DSW09]. L'impact au niveau du *gameplay* reste cependant décevant.

2.6 Et ailleurs (jeux hors catégorie)

Creatures[13] (1996, Steve Grand) : L'objectif de ce jeu est d'élever des créatures, appelées *Norns*, en interagissant directement avec chacune. Le joueur assure ainsi le rôle de professeur, de parent, chargé d'enseigner aux *Norns* les bases de la vie en société, du langage et de la survie. Chaque créature est composée d'un « cerveau » (AI) et d'un corps. L'intelligence artificielle de chaque créature est gérée par un réseau de neurones modulaire, c'est-à-dire composé de différents sous-réseaux possédant chacun son fonctionnement et son rôle propre. Cette modularité permet une efficacité accrue des processus d'apprentissage ainsi qu'une modélisation plus fine des comportements non-déterministes en réponse à l'environnement.

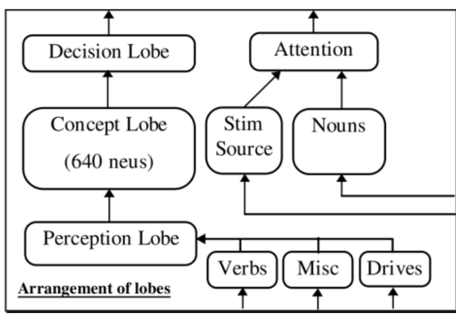

FIGURE 2.9 – Schéma des modules du réseau neuronal

Chaque réseau modulaire est attaché à un corps, lui-même régi par un génome dynamique autorisant l'hérédité et les mutations. Grâce à certains gènes dont l'expression dépend du temps, ce génome parvient à simuler « l'âge » de chaque *Norn*, et les comportements afférents. Le corps est en outre en interaction permanente avec le cerveau grâce à un système d'hormones (ordres du cerveau au corps) et de toxines ou de bactéries (informations du corps au cerveau). Enfin, des algorithmes génétiques gèrent l'évolution des corps et des cerveaux, et

13. cf. `http://aigamedev.com/open/highlights/creatures-ai/`

permettent d'obtenir une macro-dynamique des populations de *Norns* au réalisme saisissant. La modularité du cerveau et la dissociation entre ce dernier et le corps permet l'apparition de comportements émergents en réponse aux indications du joueur et à l'environnement - le monde dans lequel vivent les *Norns* possédant en effet une faune et une flore variées assurant les besoins des *Norns* mais pouvant aussi leur nuire.

Black&White (2001, Richard Evans) : Le joueur endosse le rôle d'un dieu et détermine le destin des villageois. Les créatures peuvent apprendre des comportements complexes, et le choix du bien ou du mal a des conséquences sur le déroulement ultérieur du jeu. D'après le peu d'informations que nous avons pu recueillir à ce sujet, la structure sous-jacente de l'intelligence artificielle des différentes entités du jeu repose sur des réseaux neuronaux, couplés à des algorithmes plus « classiques » - ainsi qu'à une certaine dose de modifications empiriques.

Chapitre 3

État de l'art des utilisations de *réelle* intelligence artificielle

3.1 Généralités

3.1.1 Concepts généraux

Définitions préalables

Agent un agent est une entité qui perçoit l'environnement dans lequel elle se trouve grâce à des capteurs, et agit au sein de cet environnement à l'aide d'effecteurs (cf. Figure 3.1).

Environnement l'environnement est la structure dans laquelle évolue l'agent. Il est caractérisé par un ensemble de paramètres arbitrairement nombreux, et spécifiques à chaque problème.

Percept un percept désigne les entrées (perceptions) d'un agent à un instant donné.

Action une action est le moyen par lequel l'agent peut influer sur l'environnement qui l'entoure.

État un état désigne une configuration dans lequel se trouvent l'environnement et l'agent. Il peut dépendre de paramètres liés à l'environnement dans lequel évolue l'agent ou de paramètres qui sont propres à l'agent.

Mesure de performance une mesure de performance intègre le critère de succès du comportement d'un agent. Un agent effectue, dans un environnement donné, une action ou une séquence d'actions en fonction d'un percept ou d'une séquence de percept. Ces actions influent sur l'environnement et sur les états traversés par l'agent. Si ceux-ci sont souhaitables, c'est que l'agent s'est bien comporté - le problème étant de définir le terme « souhaitable ». L'agent étant dans bien la plupart des cas incapable de donner un sens à cette expression, c'est au concepteur de l'agent de définir une mesure de performance objective.

Agent rationnel un agent rationnel doit sélectionner une action parmi les actions qu'il a à sa disposition, afin de maximiser sa mesure de performance, en tenant compte des observations fournies par ses percepts et de la connaissance dont il dispose.

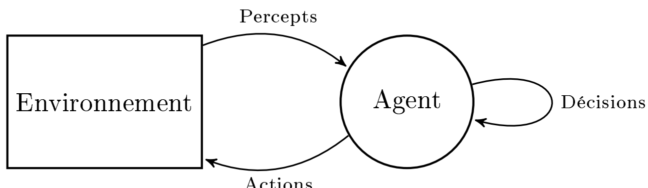

FIGURE 3.1 – Schéma basique d'un agent

Caractéristiques des environnements

Entièrement ou partiellement observable un agent peut accéder à toute l'information disponible dans l'environnement si ces capteurs le permettent : il se trouve alors dans un environnement qui est *entièrement observable*. En revanche, il est possible que l'agent soit dans l'incapacité de prendre connaissance de certains paramètres ou de parties de l'univers, auquel cas il se trouve dans un univers *partiellement observable*.

Déterministe ou stochastique si l'état à venir est complètement déterminé par l'état courant et par l'action qu'exécute l'agent, l'environnement est dit *déterministe*. Si au contraire, plusieurs états peuvent être atteints pour une même action à partir d'un même état initial, l'environnement est *stochastique*.

Épisodique ou séquentiel dans un environnement de tâche *épisodique*, l'expérience de l'agent est divisée en épisodes atomiques : à chaque épisode, l'agent perçoit puis exécute une action unique. Dans ces environnements, le choix de l'action de chaque épisode dépend uniquement de l'épisode lui-même et non pas des épisodes passés. Un automate qui doit décider de la conformité d'une pièce évolue dans un environnement épisodique. En revanche, dans un environnement séquentiel, la décision prise à l'instant présent est susceptible d'affecter les décisions futures. Ainsi, une partie d'échecs se déroule dans un environnement séquentiel.

Statique ou dynamique un environnement statique n'évolue pas alors que l'agent évolue en son sein, tandis qu'un environnement dynamique peut voir certains de ses paramètres se modifier avec les actions de l'agent.

Discret ou continu selon les variables de l'environnement considéré, ce dernier est dit discret ou continu. Par exemple, la façon dont évolue le temps peut être discrète (jeu d'échecs par exemple) ou continue (conduite automobile).

Mono- ou multi-agent cette distinction traduit simplement le fait que dans un environnement, un agent peut évoluer seul ou être confronté à d'autres agents. Une partie d'échecs se déroule dans un environnement multi-agents, alors que la résolution d'une grille de sudoku se déroule dans un environnement mono-agent.

Le cas le plus délicat à traiter est évidemment celui d'un environnement partiellement observable, stochastique, séquentiel, dynamique, continu et multi-agents.

3.1.2 Agents rationnels

La notion d'*agent rationnel*, qui revient de manière récurrente dans ce document, est essentielle dans la démarche que nous avons choisie d'adopter. Rappelons que si un *agent* est simplement une entité qui *perçoit* son environnement (généralement de manière incomplète) et *agit* sur lui en retour, un agent rationnel, lui, effectue toujours l'action la plus satisfaisante pour lui compte tenu de ses connaissances, c'est-à-dire **maximisant son espérance de bien-être à long terme** [RN03].

Il est important de noter que c'est l'*espérance* qu'il s'agit de maximiser, non le bien-être lui-même : en effet, en raison de l'incertitude inhérente à ses actions, au caractère incomplet des informations dont il dispose, et au non-déterminisme de certains aspects du monde dans lequel il évolue, un agent, tout perfectionné qu'il puisse être, ne travaille jamais qu'avec des *probabilités* de succès ou d'échec.

Différents modèles possibles

La définition ci-dessus ne préjuge en rien du fonctionnement interne de l'agent rationnel ; celui-ci est laissé absolument libre, pour peu que le résultat voulu soit atteint. Brièvement, présentons les modèles les plus courants :

Agent réflexe simple

Ce type d'agent dispose d'une liste de règles élémentaires, de la forme

if condition **then** action

Il s'agit d'une généralisation d'une approche élémentaire, qui consisterait à remplir une table *état* $\rightarrow$ *action*, approche impraticable dès lors que l'espace des états voit sa taille augmenter. L'avantage de cette généralisation est la possibilité d'ajouter dynamiquement des règles, ou d'en affiner certaines, par exemple *via* un langage de programmation dynamique [1].

Agent réflexe à modèle ou état

Cette fois, l'agent tente de surcroît de bâtir une *représentation interne du monde*, modèle qui évolue en fonction des actions essayées et des modifications de l'environnement que ces actions semblent entraîner. Ainsi, si lâcher un objet se solde à chaque fois par une évolution observable de la composante verticale de la vitesse de cet objet, un agent à modèle pourra en déduire, selon le degré de sophistication de sa représentation interne et de ses méthodes de mise à jour :

i. que les objets lâchés dans le passé ont changé de position
ii. que les objets ont tendance à s'éloigner de sa main
iii. que les objets ont tendance à se rapprocher du sol
iv. la loi générale de la gravitation

1. Voir `http://en.wikipedia.org/wiki/Dynamic_programming_language` pour plus de détails sur cette classe de langages de programmation.

Ce type d'agent garde donc *a minima* en mémoire un *état courant*, composé des actions déjà effectuées, observations et percepts précédents, etc., et ce afin d'affiner sa connaissance de l'environnement et du contexte réel de ses actions. Cet état est mis à jour à chaque percept reçu, pour refléter idéalement tant les nouvelles informations disponibles que l'évolution *probable*, compte tenu du fonctionnement du monde, des parties de l'environnement non directement observables - la prise de décision est dès lors - si le modèle est bon - bien plus précise que dans le cas précédent.

Agent à buts

Une connaissance accrue du monde dans lequel l'agent évolue n'est pas toujours suffisante : il est souvent nécessaire, pour décider de l'action adéquate, de prendre en compte *ce que l'on cherche à faire*. C'est le principe qui sous-tend les agents à buts, qui, en plus d'apprendre au fur et à mesure ce que leurs actes impliquent, combinent cela à un *critère de satisfaction de leurs objectifs* (la Figure 3.2 présente schématiquement le fonctionnement d'un tel agent). Il devient possible, dès lors, d'envisager des planifications et stratégies à long terme - ce que ne permettaient pas les deux types d'agent précédents.

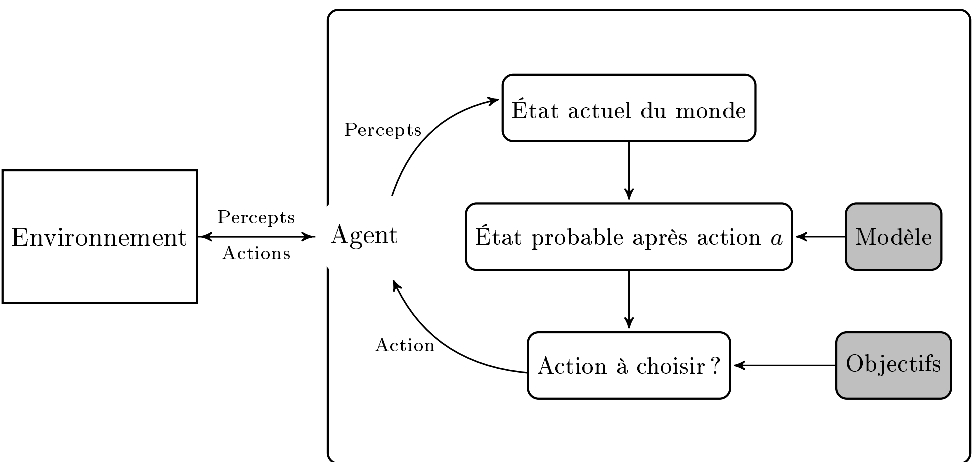

FIGURE 3.2 – Schéma d'un agent à buts

Agent à utilité

Dans bien des cas, plusieurs stratégies permettront de mener à bien un objectif donné ; cependant, l'une se révélera bien plus « satisfaisante » que l'autre (eg, moins longue, ou moins coûteuse ...). Les agents à buts souffrent donc d'un certain manque de *précision* dans leur approche ; c'est ce manque que visent à pallier les agents à utilité [2]. *La plus grande difficulté est donc de choisir (ou construire, dans le cadre d'un agent capable d'apprentissage), une fonction d'utilité pertinente*, c'est-à-dire reflétant bien le bien-être ou la satisfaction réels apporté par un état donné.

2. Si l'on considère que le critère d'atteinte d'objectif d'un agent à buts est une fonction $f\colon E \to \{0,1\}$, alors un agent à utilité peut être vu comme disposant d'une fonction similaire $u\colon E \to [0;1]$.

Tous ces types d'agents peuvent être dotés d'une *capacité d'apprentissage*, c'est-à-dire une aptitude à modifier leurs critères ou structures décisionnels au fur et à mesure qu'ils engrangent de l'expérience, ceci à l'aide de la mesure de performance (donnée externe à l'agent) de l'environnement.

3.2 Réseaux de neurones

Inspirés par les recherches en psychologie et en neurophysiologie du début du XXe siècle concernant le fonctionnement du cerveau et plus généralement les mécanismes de la pensée, les *réseaux neuronaux artificiels* furent formalisés dans les années 1950, alors que l'engouement autour des machines « intelligentes » battait son plein. Alors que l'ordinateur balbutiait encore, on se demandait déjà s'il pourrait penser, agir comme un être humain ; à tout le moins cherchait-on un moyen pour résoudre automatiquement des problèmes triviaux pour un cerveau, comme la reconnaissance de caractères ou la classification. Grâce à la neurobiologie, on s'aperçut vite que la puissance du cerveau résidait dans son fonctionnement dynamique et parallèle. Les réseaux de neurones, dont la forme la plus simple, le *perceptron*, fut dévoilée en 1957 par F. Rosenblatt, sont une modélisation de cette structure évolutive. Nous présenterons ici la formalisation mathématique de tels réseaux et étudierons l'utilisation de différents algorithmes d'« éducation » des réseaux, avant de considérer leurs applications potentielles dans le cadre de notre projet.

3.2.1 Description et formalisation mathématique

Les *réseaux de neurones* sont des cas particuliers de graphes orientés, dans lesquels les nœuds sont des calculateurs élémentaires - les neurones - reliés entre eux par des arêtes pondérées. Formellement, un neurone est la donnée d'une fonction $f : \mathbb{R} \to \mathbb{R}$ et d'un *biais* $b \in \mathbb{R}$ tel que, soumis à un ensemble d'entrées $\vec{x}$ pondérées par le *vecteur de poids* $\vec{w}$, il renvoie :

$$s = f(\vec{w}^{\mathrm{T}}\vec{x} - b)$$

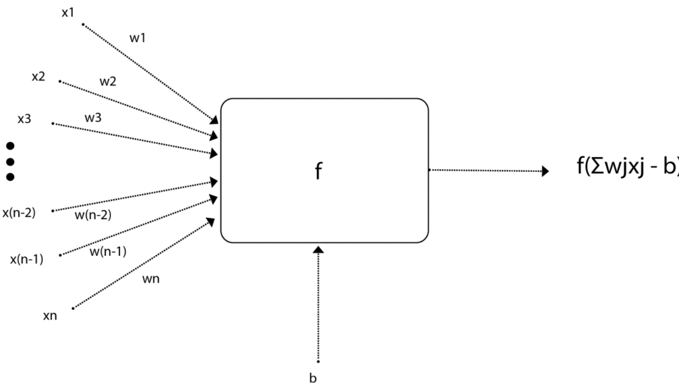

FIGURE 3.3 – Entrées et sortie d'un neurone

C'est à partir de cette brique élémentaire (voir Figure 3.3) qu'on définit le réseau de neurones : il s'agit d'un ensemble de neurones regroupés en *couches*, une couche contenant plusieurs neurones[3] soumis aux mêmes entrées et possédant la même fonction de transfert - l'entrée de la couche $k+1$ étant la sortie de la couche k. On définit :

- $\vec{p}$ le vecteur d'entrée du réseau
- $W^{(k)} = [w^k_{i,j}]$ la matrice des poids de la k^e couche (avec $w^k_{i,j}$, abrégé en $w_{i,j}$, le poids de l'arête liant le i^e neurone de la couche à la j^e entrée)
- $F^{(k)}$ la fonction de transfert de la k^e couche (telle que $F^{(k)}(\vec{x}) = \left(f^{(k)}(x_1), \ldots, f^{(k)}(x_n)\right)^{\mathrm{T}}$)
- $\vec{b}^{(k)}$ le vecteur des biais de la k^e couche.

Soit N le nombre de couches et $\vec{s}$ la sortie du réseau. On a alors :

$$\vec{s} = F^{(N)}\left(W^{(N)} \cdot F^{(N-1)}\Big(W^{(N-1)} \cdot F^{(N-2)}\big(\ldots(F^{(1)}(W^{(1)} \cdot \vec{p} - \vec{b}^{(1)}))\ldots\big) - \vec{b}^{(N-1)}\Big) - \vec{b}^{(N)}\right)$$

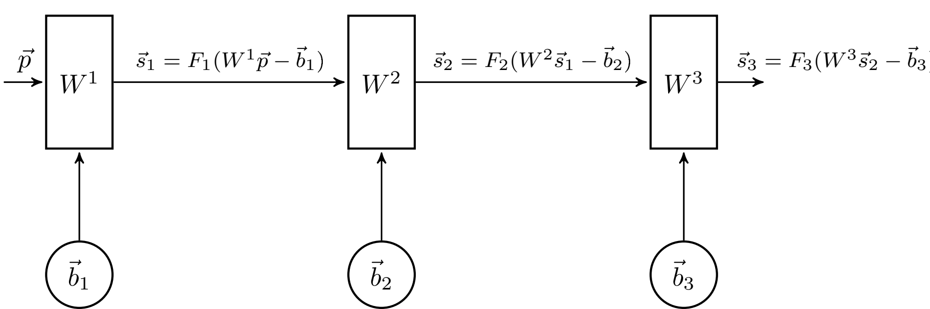

FIGURE 3.4 – Schéma d'un réseau multicouche

Les fonctions de transfert utilisées varient d'une application à l'autre, mais font la plupart de temps partie d'un ensemble de fonctions standards, regroupées (non exhaustivement) dans le tableau ci-dessous. Remarquons toutefois que le choix d'au moins une fonction non-linéaire pour le réseau est judicieux, ce dernier étant lui-même en mesure, quitte à augmenter sa taille, d'émuler n'importe quelle fonction linéaire.

3. Les différentes couches ne comportant pas nécessairement le même nombre de neurones.

Nom de la fonction	Expression	Graphe
Seuil (Heaviside [4])	$f(x) = \begin{cases} 0 & \text{si} \quad x < 0 \\ 1 & \text{si} \quad x \geq 0 \end{cases}$	
Seuil symétrique [5]	$f(x) = \begin{cases} -1 & \text{si} \quad x < 0 \\ 1 & \text{si} \quad x \geq 0 \end{cases}$	
Linéaire	$f(x) = x$	
Sigmoïde	$f(x) = \frac{1}{1+\mathrm{e}^{-x}}$	

Le réseau de neurones le plus classique, que nous serons amenés à utiliser au sein de notre projet, en raison justement de la richesse des résultats théoriques existants à son sujet, est le *perceptron*. Il peut être simple (une seule couche) ou multicouche, et se caractérise par l'utilisation de la fonction seuil symétrique pour toutes les fonctions de transfert. Analysons tout d'abord le comportement d'un neurone du perceptron, de vecteur de poids $\vec{w}$: lorsqu'une entrée $\vec{x}$ lui est soumise, le neurone renvoie

$$s = \begin{cases} 1 & \text{si} \quad \vec{w}^{\mathrm{T}}\vec{x} - b = \sum_j^N w_j x_j - b \geq 0 \\ -1 & \text{sinon} \end{cases}$$

Ce neurone sépare donc l'espace des entrées en deux demi-espaces, dont l'hyperplan séparateur a pour équation : $\sum_{j=1}^n w_j x_j = b$

Lorsqu'on considère une couche formée de plusieurs neurones de ce type, on constate donc que, chaque neurone séparant l'espace par un hyperplan différent (en général), on est capable de partitionner l'espace des entrées en fonction de l'appartenance d'une entrée à l'un des demi-espaces du neurone j, ceci pour tous les neurones de la couche. Ce « quadrillage » de l'espace des entrées par des hyperplans a cependant une limite : seuls les problèmes linéaires peuvent être résolus grâce à un perceptron simple.

On admettra que le *perceptron multicouche* permet quant à lui de s'affranchir de la limite de linéarité du problème en engendrant des frontières concaves ou convexes, ouvertes ou fermées de complexité arbitraire (pour un réseau de

4. Du nom de Oliver Heaviside, physicien britannique du XIXe siècle

5. Dans certains cas pratiques, la fonction de seuil symétrique est remplacée par une sigmoïde bien choisie, afin d'éviter la divergence de la dérivée en 0. Peu de résultats théoriques existent concernant les réseaux dont les fonctions de transfert sont des sigmoïdes, mais l'expérience prouve que ces derniers ont un comportement très proche de celui des perceptrons.

3 couches), au détriment cependant de l'efficacité des algorithmes d'entraînement [Par04].

Comme nous l'avons mentionné en introduction, la puissance des réseaux neuronaux réside dans leur capacité à apprendre à **résoudre un problème de la meilleure façon possible**. Cet apprentissage passe par l'utilisation de diverses règles et algorithmes, que nous allons présenter.

3.2.2 Processus d'apprentissage

Comme on l'a vu dans la première partie, la réponse d'un réseau de neurones fixé à une entrée est déterminée par les valeurs des poids. Le processus d'apprentissage pour un réseau consiste donc dans la mise en place de règles dynamiques et itératives d'*ajustements des poids*, afin que le réseau donne la solution du problème pour lequel on souhaite l'entraîner, ou au moins une bonne approximation de cette solution. Les propriétés « dynamiques » et « itératives » du processus d'apprentissage font la force du réseau neuronal, car lui permettent d'évoluer, d'apprendre et de progresser au contact du problème.

On distingue deux grands paradigmes d'apprentissage :

- L'**apprentissage supervisé** : le réseau peut dans ce cas fonctionner soit en apprentissage, soit en « production ». La phase d'apprentissage consiste dans la soumission au réseau d'entrées associées aux sorties attendues. A l'aide de ces « patrons », le réseau ajuste ses poids afin de faire coïncider au mieux les valeurs qu'il renvoie avec les sorties attendues. Pendant la phase de production, en revanche, les poids n'évoluent plus et le réseau est utilisé pour résoudre le problème souhaité, étant alors confronté à des entrées qu'il n'a *a priori* pas rencontrées pendant l'apprentissage.
- L'**apprentissage non supervisé** : les phases d'apprentissage et de production sont simultanées. Le réseau possède alors un « indicateur de qualité » [6] des sorties, qu'il utilise en permanence afin d'ajuster ses poids en temps réel.

Comme nous le verrons dans la troisième partie, ces deux possibilités seront utilisées dans notre projet en fonction de l'utilisation du réseau.

Dans le cas de l'apprentissage supervisé, il est souvent nécessaire de définir un critère dit d'arrêt qui permet de déterminer la fin de l'entraînement (critère d'entraînement « suffisant »). En effet, rien ne garantit le temps de convergence des algorithmes.

Outre cette principale distinction, il existe différentes règles d'ajustement des poids afin d'entraîner le réseau. Nous présentons ici trois de ces règles : la *méthode de correction d'erreur*, utilisable en apprentissage supervisé pour un réseau simple, la *méthode de Hebb*, utilisable quel que soit le paradigme d'apprentissage, et la *méthode de rétropropagation des erreurs*, similaire à la correction d'erreur, mais utilisable pour des réseaux multicouches.

6. Une fonction de *feedback* évaluant, d'après des critères fixes, la pertinence (performance) de la sortie qu'il a obtenue)

Correction d'erreur

Cette règle est une méthode d'apprentissage supervisé. Soit donc $\left\{(\vec{p}_1, \vec{d}_1), \ldots, (\vec{p}_n, \vec{d}_n)\right\}$ un ensemble de couples *(entrée, sortie attendue)*. La méthode de correction d'erreur consiste à minimiser l'écart quadratique entre sorties réelles et sorties attendues. Cette erreur quadratique vaut par définition :

$$\langle \varepsilon^2 \rangle = \frac{1}{n} \sum_{k=1}^{n} \left(\vec{d}_k - \vec{s}_k \right)^2$$

On cherche alors à déterminer $\Delta W = W(t+1) - W(t)$ tel que $\langle \varepsilon_k^2 \rangle$ soit minimal (où W est la matrice des poids de la couche du réseau simple et t représente le cycle d'apprentissage, i.e. est la matrice des poids après un entraînement du réseau au stade t). On démontre alors que la direction de descente optimale est celle donnée par le gradient de $\langle \varepsilon_k^2 \rangle$:

$$\Delta W = -\eta \nabla_W \langle \varepsilon_k^2 \rangle$$

Où η est une constante positive, appelée *taux d'apprentissage*. Cette règle est aussi appelée « descente du gradient ». Elle dépend des fonctions de transfert choisies. Nous détaillerons plus avant cette méthode lorsque nous décrirons la méthode de rétropropagation des erreurs, p.32.

Méthode de Hebb

Cette méthode d'apprentissage non-supervisé s'inspire très largement des travaux du neurophysiologiste Donald Hebb [7], et repose sur une adaptation dynamique des poids en fonction des activités pré- et post-neuronales.
On peut modéliser cette règle sous la forme :

$$\Delta w = \eta \vec{x} \cdot \vec{s} - \alpha w(t-1)$$

où w désigne le poids associé à l'arête entre un neurone donné et l'entrée $\vec{x}$ sur ce neurone, $\vec{s}$ désigne la sortie du neurone, et η, α sont deux constantes telles que $|\alpha| \leq 1$.

Le terme principal, en $\eta \vec{x} \cdot \vec{s}$, dépend à la fois de l'entrée et de la sortie du neurone, et quantifie la corrélation entre les deux. Il simule le modèle physiologique de Hebb, qui postule qu'une liaison synaptique entre deux neurones se renforce lorsqu'elle est beaucoup utilisée, et est inhibée dans le cas contraire (si les neurones qu'elle lie sont par exemple activés de façon asynchrone) [8].

Le terme en $\alpha w(t-1)$ est lui un terme « mémoire » permettant d'éviter des problèmes de divergence (si par exemple $\vec{x}$ et $\vec{s}$ sont constants). Les deux constantes sont à ajuster en fonction du problème, en remarquant toutefois que le signe de η détermine si le réseau évolue « pour » ou « contre » le résultat qu'il a renvoyé.

---

7. Donald Hebb (1904-1985) est considéré comme l'un des pères de la neuropsychologie et des réseaux neuronaux (`http://users.ecs.soton.ac.uk/harnad/Archive/hebb.html`)

8. « When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic changes take place in one or both cells such that A's efficiency as one of the cells firing B, is increased. »
(Hebb, *The Organization of Behavior : A Neuropsychological Theory*)

Dans le cadre de notre projet, nous n'allons pas appliquer directement cette méthode, mais plutôt nous en inspirer afin de simuler un processus de réponse instantanée aux actions et aux résultats de ces actions dans le jeu. Nous détaillerons cette application dans le chapitre 4.

Rétropropagation des erreurs

Cette méthode est une généralisation de la méthode de correction d'erreur au cas des réseaux multicouches. On dispose toujours d'un ensemble $\{(\vec{p}_1, \vec{d}_1), \ldots, (\vec{p}_n, \vec{d}_n)\}$ de couples (entrée, sortie attendue) ; cependant, cela ne nous donne d'information directe que sur la couche de sortie. Par conséquent, pour ajuster les autres poids, il est nécessaire de faire « remonter » l'information. Qualitativement, on cherche à *rétro-propager l'erreur aux couches en amont*, en n'attribuant à chaque couche qu'une portion de l'erreur totale, portion relative à la contribution de la couche à la sortie. Cette attribution relative permet une sorte de « spécialisation » des neurones, chaque neurone étant entraîné à reconnaître une caractéristique de l'espace des entrées [FS91].
Formalisons maintenant la méthode.

On cherche à minimiser l'erreur quadratique pour un couple $(\vec{p}_l, \vec{d}_l)$:

$$\varepsilon_l^2 = \left(\vec{d}_l - \vec{s}_l\right)^2$$

Pour cela, on applique une « descente du gradient », généralisée aux biais :

$$\Delta w_{i,j}^k = -\eta \frac{\partial \varepsilon_l^2}{\partial w_{i,j}^k}$$
$$\Delta b_i^k = -\eta \frac{\partial \varepsilon_l^2}{\partial b_i^k}$$

pour la k^e couche.

On calcule alors

$$\frac{\partial \varepsilon_l^2}{\partial w_{i,j}^k} = \frac{\partial \varepsilon_l^2}{\partial a_i^k} \frac{\partial a_i^k}{\partial w_{i,j}^k}$$
$$\frac{\partial \varepsilon_l^2}{\partial b_i^k} = \frac{\partial \varepsilon_l^2}{\partial a_i^k} \frac{\partial a_i^k}{\partial b_i^k}$$

avec $a_i^k = \sum_{j \in \{\text{entrées de la } k^e \text{ couche}\}} w_{i,j}^k s_j^{k-1} - b_i^k$ le niveau d'activation du neurone.
Notons $\sigma_i^k = \frac{\partial \varepsilon_l^2}{\partial a_i^k}$ la sensibilité de ε_l^2 aux changements dans le niveau d'activation du neurone i de la couche k, et $\vec{\sigma}_k = (\sigma_i^k)_i$. On obtient alors :

$$\frac{\partial \varepsilon_l^2}{\partial w_{i,j}^k} = \sigma_i^k s_j^{k-1}$$
$$\frac{\partial \varepsilon_l^2}{\partial b_i^k} = -\sigma_i^k$$

ce qui donne, en notation matricielle :

$$\Delta W^k = -\eta \vec{\sigma}_k \vec{s}_k^{\mathrm{T}}$$
$$\Delta \vec{b}^k = \eta \vec{\sigma}_k$$

Il reste à déterminer les sensibilités $\vec{\sigma}_k$. Par souci de clarté, on admettra ici le calcul, réalisé dans la référence [Par04]. Ce calcul repose sur le fait qu'on peut relier $\vec{a}_{k+1}$ à $\vec{a}_k$ grâce à la fonction de transfert de la couche k, établissant ainsi une relation de récurrence entre $\vec{\sigma}_{k+1}$ et $\vec{\sigma}_k$:

$$\vec{\sigma}_k = \dot{F}^k(\vec{a}_k) \left(W^{k+1}\right)^{\mathrm{T}} \cdot \vec{\sigma}_{k+1}$$

où $\dot{F}^k(\vec{a}_k) = \mathrm{Diag}\Big(\dot{f}^k(a_i^k)\Big)_i$ et $\dot{f}^k(a_i^k) = \frac{\partial f^k}{\partial a_i^k}(a_i^k)$.

Sachant de plus que $\vec{\sigma}_N = -2\dot{F}^N(\vec{a}_N)(\vec{p}_k, \vec{d}_k)$, où N est le nombre de couches du réseau, il devient possible de calculer de proche en proche $\vec{\sigma}_{N-1}, \ldots, \vec{\sigma}_1$.

L'algorithme peut donc être résumé ainsi :

Input : $\{(\vec{p}_1, \vec{d}_1), \ldots, (\vec{p}_n, \vec{d}_n)\}$ {Couples (entrée, sortie attendue)}
Ensure: $\forall k$, W^k est optimale {Ajuste les poids du réseau de neurones}
1: **for all** k **do**
2: $W^k \leftarrow$ `RANDOM_MATRIX()` {Initialiser les poids à des valeurs aléatoires}
3: **end for**
4: **for all** $(\vec{p}_l, \vec{d}_l)$ **do**
5: Propager l'entrée $\vec{p}_l$ dans le réseau jusqu'à la sortie {Calcul des $\vec{s}_k$, $\vec{a}_k$}
6: Rétropropager les sensibilités vers l'arrière {Calcul des $\vec{\sigma}_k$}
7: Mettre à jour les poids et les biais {Méthode du gradient}
8: **end for**
9: **if** `CRITERE_ARRET()` **then**
10: **return** $(W^k)_k$
11: **else**
12: Permuter l'ordre de présentation des couples $(\vec{p}_l, \vec{d}_l)$
13: Retourner L. 4
14: **end if**

Remarque Un critère d'arrêt courant est la comparaison de l'erreur quadratique moyenne à une constante δ arbitrairement proche de 0 :

$$\langle \varepsilon^2 \rangle = \frac{1}{n} \sum_{k=1}^{n} \left(\vec{d}_k - \vec{s}_k\right)^2 < \delta$$

Nous constatons donc qu'entraîner un réseau de neurones peut être un processus coûteux en particulier lorsque le nombre de couches augmente. En outre, l'expérience prouve qu'augmenter le nombre de couches peut parfois *réduire* la qualité des réponses du réseau [9]. Il faut donc estimer pour chaque application

9. Le réseau, *trop* précis, modélise même le bruit inhérent aux couples fournis pour l'apprentissage.

le nombre de couches nécessaires, les fonctions de transfert, et décider de l'algorithme d'entraînement à utiliser.

3.3 Algorithmes génétiques

3.3.1 Une analogie avec la biologie

Les algorithmes génétiques visent à trouver une bonne solution à un problème (mais pas forcément la meilleure) en s'inspirant des théories de la biologie sur l'évolution des espèces.

En biologie, les caractéristiques d'un individu sont codées par son matériel génétique (gènes, chromosomes), et influent sur son adaptation à un milieu. Les individus les mieux adaptés au milieu de vie ont plus de chances de survivre, et donc de se reproduire, que les autres. Ainsi les gènes codant des caractéristiques qui avantagent l'individu ont tendance à se répandre dans les générations suivantes, tandis que les gènes codant pour des caractéristiques moins avantageuses ou handicapantes tendent à disparaître.

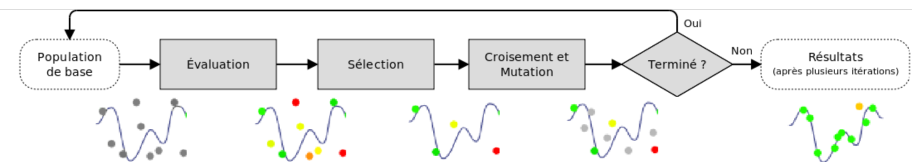

FIGURE 3.5 – Schéma récapitulatif d'un algorithme génétique

Le même principe est appliqué à la résolution d'un problème. On part d'une population de solutions à un problème, auxquelles on attribue une note en fonction de leur efficacité : c'est l'évaluation. On procède ensuite à différentes opérations :

Sélection Par analogie avec la sélection naturelle, on pratique une sélection artificielle parmi les solutions pour délaisser les moins efficaces.

Croisements À l'instar des chromosomes, qui s'échangent des gènes lors d'enjambements (ou recombinaisons) au cours de la reproduction, les solutions sélectionnées ont une certaine *probabilité de se recombiner* pour former de nouvelles solutions, possédant des caractéristiques des deux solutions mères. On opte généralement pour un taux de croisement l'ordre de 0.8.

Mutations Elles apparaissent de façon aléatoire lors de la reproduction du génome et consistent en la *substitution d'un gène par un autre, de manière imprévisible*. Ainsi, une solution sélectionnée a une probabilité non nulle d'être spontanément modifiée, de façon aléatoire. Le taux de mutation est généralement faible (entre 0.01 et 0.001) pour ne pas rendre l'algorithme trop aléatoire. La mutation sert à éviter une convergence prématurée de l'algorithme ; par exemple, dans le cas de la recherche d'extremum, d'éviter la convergence vers un extremum local [10]

10. Cela peut être vu comme une perturbation aléatoire, qui permet de "sortir" d'un min-

3.3.2 Mise en œuvre

Évaluation

L'évaluation consiste à estimer à quel point une solution donnée répond au problème. Il s'agit de calculer ce qu'on appelle le *fitness* de la solution. Pour la comparaison d'images par exemple, on peut compter le nombre de pixels semblables, ou bien sommer les écarts entre les valeurs RGB du pixel original et du pixel reproduit ; pour la vie artificielle, utiliser la quantité moyenne de nourriture ingérée, ou la durée de survie.

De manière générale, lorsqu'on demande à une solution P d'approcher une fonction qui, pour une valeur e_i donnée, renvoie s_i ($i \in [0, N]$), on calcule le fitness ϕ de S à f :

Par exemple comme somme des valeurs absolues des écarts :

$$\phi = \sum_{i=1}^{n} |P(e_i) - s_i|$$

Ou bien en calculant l'erreur quadratique :

$$\phi = \sum_{i=1}^{n} (P(e_i) - s_i)^2$$

Le but étant alors de minimiser le *fitness*. Les principales formes de *fitness* sont :

Fitness standardisé : la valeur du meilleur *fitness* possible est 0, toutes les valeurs de *fitness* sont positives.

Fitness normalisé : la valeur du *fitness* est toujours comprise entre 0 et 1.

Fitness ajusté : un *fitness* normalisé où le meilleur score possible est 1.

Sélection

Il existe plusieurs techniques de sélection, dont notamment :

- **Sélection par rang** : les solutions possédant les meilleures notes sont conservées. Aucun hasard n'entre ici en compte : on choisit juste les k solutions les plus efficaces.
- **Sélection proportionnelle** (ou *Roue de la fortune*) : la probabilité pour chaque solution d'être retenue est proportionnelle à sa note. On représente les solutions sur une roue par une portion dont l'aire est proportionnelle à leur score ; un tirage homogène est ensuite effectué sur cette roue.
- **Sélection par tournoi** : on effectue (ou non) une sélection proportionnelle sur des paires alétoires solutions, puis on choisit la solution la plus efficace de chaque paire.
- **Sélection uniforme** : un tirage aléatoire des solutions à conserver, quelles que soient leurs notes respectives.

imum local pour continuer la recherche dans son voisinage - et potentiellement de converger ailleurs, vers un minimum global dont on ne pourra plus s'éloigner.

Croisement

Pour croiser deux solutions, il est important de savoir comment les représenter. Il existe principalement trois types de représentation :

Codage binaire : les solutions sont codées par une suite de bits. Historiquement le premier codage utilisé, la plupart des théories génétiques prouvent son efficacité et il reste encore majoritaire de nos jours. L'argument majeur est qu'un code binaire est une chaîne de caractères plus longue que tout autre type de code, augmentant ainsi les manières dont deux solutions peuvent se recombiner.

Codage à caractère multiple : les solutions sont codées non plus par des chaînes de bits, mais de caractères (lettres, chiffres . . .). Ce type de codage est plus naturel (cf. section 3.3.3).

Codage sous forme d'arbre : très utilisé pour coder un comportement, il représente la solution sous forme d'un arbre dont les feuilles sont des actions et les nœuds des conditions, des opérateurs, etc. Il permet de ne pas limiter la taille des solutions : n'importe quelle taille d'arbre peut être obtenue en recombinant les arbres existants entre eux (cf. Figure 3.6). Le problème est justement que l'on peut voir apparaître des solutions de très grande taille, qui ralentissent la vitesse d'exécution et qui ne sont pas forcément meilleures que certaines solutions de petite taille.

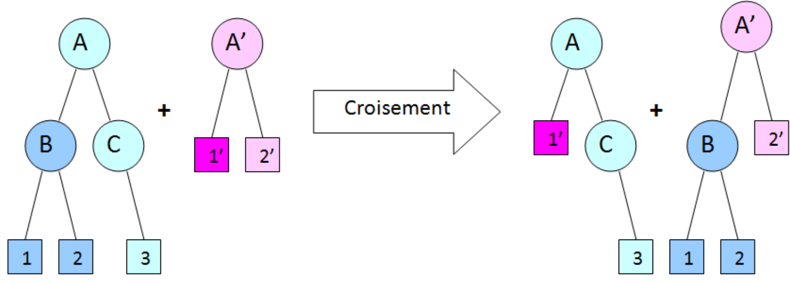

FIGURE 3.6 – Croisement de deux arbres : échange de sous-arbres

3.3.3 Applications actuelles

Problèmes de parcours de graphes

Dans la recherche, ce genre d'algorithme est notamment utilisé pour le problème dit *du voyageur de commerce*, pour lequel il n'existe pas de méthode de résolution exacte en un temps raisonnable [11]. Il est en effet assez facile de l'appliquer à la recherche d'un plus court chemin dans un graphe, et les résultats trouvés sont relativement corrects.
La fonction d'évaluation est alors la longueur du chemin, et le « gène » est la liste chronologique des nœuds à parcourir (étiquetés par des numéros).

11. Le problème du voyageur de commerce *(Travelling Salesman Problem)* consiste, étant donné un graphe pondéré, à trouver un cycle de coût ("distance") total minimal passant

Chemin	Codage
A	1234|56789
B	958342716
Fils	123495876

Table 3.1 – Croisement entre deux chemins

Pour croiser deux solutions, on recopie les nœuds de la première jusqu'à une « cassure » ; puis on recopie ceux de la seconde qui n'ont pas encore été parcourus (voir Tableau 3.1 ; la cassure est indiquée par le caractère "|").

L'entreprise Motorola s'est servie d'algorithmes génétiques pour réaliser des tests de leurs programmes. En effet, un tel test n'est valable que si *toutes* les sous-fonctions du programme ont été appelées, et tous les cas de figure possibles envisagés : cela équivaut à parcourir un graphe constitué de ces cas de figure. Le problème qui se pose est donc celui de parcourir l'intégralité du graphe ; Motorola a utilisé des algorithmes génétiques pour concevoir des fonctions tests adaptées à leurs différents programmes.
Enfin, la NASA a également fait usage de ces algorithmes pour la recherche d'itinéraire lors de la mission d'exploration de Mars du robot Pathfinder.

Optimisation dans l'industrie

Des algorithmes génétiques sont employés en aérodynamique, pour améliorer le profil des ailes d'avions notamment, en vue de réduire leur critère de "bang" sonique tout en optimisant les coefficients aérodynamiques (trainée et portance) [ODD03].
L'*optimisation structurelle* fait aussi appel à de tels algorithmes, pour minimiser le poids d'une structure en tenant compte des contraintes auxquelles elle doit résister. Enfin, citons l'apprentissage de la marche par le robot Aibo de Sony, entièrement accompli par programmation génétique.

Économie

Les algorithmes génétiques sont utilisés en économie depuis une quinzaine d'années. Ils servent entre autres à la résolution numérique d'équations, ainsi qu'en économétrie (modélisation, prévision d'événements rares, recherche de la forme fonctionnelle d'une série de données...) et en finance (optimisation de portefeuilles pour approcher l'indice DAX [12], prévision de performance des actions et quantification des risques ...) [RF07], [VY01].

3.4 Apprentissage par renforcement

3.4.1 Principe

L'apprentissage par renforcement s'intéresse à la façon dont un agent peut apprendre les actions à effectuer dans un environnement donné, sans la présence d'une sorte de professeur qui lui dirait ce qu'il est bien de faire dans chaque

exactement une fois par chaque nœud ("ville"). Il s'agit d'un problème NP-Complet.

12. *Deutscher AktienindeX* : principal indice boursier allemand.

circonstance. Par exemple, un agent peut apprendre à jouer aux échecs en analysant de nombreux exemples, qui sont justement fournis par le professeur. Mais que peut donc faire l'agent si plus personne ne lui fournit d'exemples ? Le problème est le suivant : *comment l'agent peut-il décider d'effectuer telle action plutôt qu'une autre, alors qu'il n'a aucun moyen de savoir ce qui est bon ou ce qui est mauvais pour lui.* Il faut donc qu'il sache d'une certaine manière que l'action qu'il a entreprise était bonne ou mauvaise - c'est là que le concept de *récompense* intervient.

L'obtention des récompenses peut s'effectuer de différentes manières selon le type de jeu considéré : gagnées à chaque point lors d'un match de tennis, à la fin d'une partie d'échecs, etc. Grâce à l'apprentissage par renforcement, un agent peut, en considérant un grand nombre de cas, estimer au mieux une fonction d'évaluation qui est la plus susceptible de représenter l'évolution du jeu pour chaque état.

3.4.2 Apprentissage par renforcement passif

On se place ici dans le cas d'un univers entièrement observable. Nous considérons un agent X dont la stratégie est déjà entièrement déterminée, ou plutôt n'est pas susceptible de changer en cours de jeu - c'est-à-dire l'action a effectuée dans un état s donné ne dépend que de l'état en question, et pas du moment auquel cet état est atteint : $a = \pi(s)$, π étant la stratégie (ou *politique*).

Nous définissons le concept de *processus décisionnel de Markov*, ou PDM, défini par trois paramètres :

i. un *état initial* $s_0 \in E$

ii. un *modèle de transition* $T\colon E \times A \times E \to [0,1]$, où $T(s,a,s')$ est la probabilité de passer de s à s' en effectuant a

iii. un *fonction de récompense* $R\colon E \to \mathbb{R}$

Le but de l'apprentissage passif est d'évaluer la performance de la stratégie π utilisée, sans toutefois connaître le modèle de transition $T(s,a,s')$ ni la fonction de récompense $R(s)$. Cela passe par la détermination de la *fonction d'utilité* que nous définissons comme suit :

$$U^\pi(s) = \mathrm{E}\Big[\sum_{t=0}^{+\infty} \gamma^t R(s_t) \,\Big|\, \pi, s_0 = s\Big] \tag{3.1}$$

où $0 < \gamma \leq 1$ est le *facteur d'escompte*, qui permet de donner plus d'importance, si l'on souhaite, aux premières actions, et s_0 est l'état initial.

Effectivement, si γ est proche de 0, on considère les récompenses appartenant à un avenir distant ($\gamma^t \simeq 0$) comme insignifiantes, tandis que les récompenses immédiates présentent en comparaison beaucoup d'importance. En revanche, pour $\gamma = 1$, on attribue autant d'importance à chacune des récompenses, présentes ou futures.

Justifions à présent la définition de l'utilité, c'est-à-dire l'existence de l'espérance de cette série infinie. Nous considérons essentiellement deux cas pouvant se présenter :

i. Les récompenses sont bornées par R_{max}, et $\gamma < 1$. Dans ce cas,

$$\sum_{t=0}^{+\infty} \left|\gamma^t R(s_t)\right| < \sum_{t=0}^{+\infty} \gamma^t R_{max} = \frac{R_{max}}{1-\gamma}$$

ii. L'agent X est garanti d'atteindre à un moment de la partie $t < +\infty$ un état terminal qui clôt le jeu. Dans ce cas, les sommes infinies sont à support fini, et $\gamma = 1$ est encore acceptable.

Il faudra faire attention à ne pas utiliser de stratégie n'aboutissant pas nécessairement à un état terminal avec un facteur d'escompte égal à 1, au risque de provoquer l'échec de l'algorithme. C'est un argument en faveur de l'utilisation des récompenses escomptées.

Nous chercherons donc à déterminer la *politique optimale* :

$$\pi^* = \underset{\pi}{\text{argmax}}\, \text{E}\Big[\sum_{t=0}^{+\infty} \gamma^t R(s_t) \,\Big|\, \pi \Big]$$

Et noterons :

$$U(s) = U^{\pi^*}(s)$$

Estimation directe de l'utilité

On considère ici que l'*utilité* d'un état est la récompense totale espérée à partir de cet état, et que chaque essai fournit un échantillon de cette valeur pour chaque état visité. Ainsi, pour un nombre infini d'essais, la loi des grands nombres assure que la moyenne convergera vers l'espérance théorique de l'équation.

Intéressons-nous, en guise d'exemple, à un agent X se déplaçant dans un carré de quatre cases (identifiées par leurs coordonnées (x, y)) :

$(1,2)$	$(2,2)$
$(1,1)$	$(2,1)$

X se trouve initialement en bas à gauche, son objectif étant d'arriver dans le coin supérieur droit. On attribue -0.05 à X s'il arrive dans une case qui n'est pas l'objectif, et $+1$ s'il atteint la case en haut à droite. Considérons la séquence de jeu suivante :

$$(1,1)\ ; (1,2)\ ; (1,1)\ ; (1,2)\ ; (2,2)$$

On obtient deux échantillons pour l'état $(1,1)$ [13], à savoir : $-0.05 \times 3 + 1 = 0.85$ et $-0.05 \times 1 + 1 = 0.95$. On met à jour l'utilité empirique de $(1,1)$ à l'aide d'une moyenne "mobile" gardée en mémoire entre chaque séquence. L'estimation directe de l'utilité réduit le problème de l'apprentissage par renforcement à un problème d'apprentissage inductif - cependant, elle néglige le fait que les utilités

13. Selon que l'on calcule l'espérance en partant de $s_0 = s_{t=0}$ ou de $s_0 = s_{t=2}$.

des états ne sont pas indépendantes. On a en effet (ce que l'agent ne sait pas, car il ne connaît pas le modèle de décision de Markov) :

$$U^\pi(s) = R(s) + \gamma \sum_{s'} T\left(s, \pi(s), s'\right) U^\pi(s') \tag{3.2}$$

relation connue sous le nom d'*équation de Bellman*.

Programmation dynamique adaptative

La *programmation dynamique adaptative* (ou PDA) cherche à tenir compte des relations entre les états, contrairement à l'estimation directe de l'utilité. L'agent essaie ici d'apprendre le modèle de transition de l'environnement à mesure qu'il progresse. Ainsi, il apprend au fur et à mesure le modèle de transition $T(s, \pi(s), s')$ et les récompenses $R(s)$, les insère dans l'équation de Bellman, et calcule ainsi les utilités des états. On représente le modèle de transitions par une table de probabilités, et l'on mémorise le nombre de réalisations de chaque résultat afin d'estimer la probabilité de transition $T(s, a, s')$ à partir de la fréquence à laquelle s' est atteint lorsqu'on exécute a dans l'état s (voir Listing 4 pour une implémentation en pseudocode).

Apprentissage par différence temporelle

L'idée de base de la différence temporelle est de définir d'abord les conditions qui sont vraies *localement* lorsque les estimations des utilités sont correctes, puis à écrire une équation de mise à jour qui fait tendre les estimations vers un équilibre.

On considère de nouveau le cas de l'agent X dans son carré de quatre cases. On considère la séquence :

$$(1,1)\ ; (1,2)\ ; (2,1)\ ; (1,1)\ ; (2,2)$$

On s'intéresse à la transition de $(1,2)$ à $(2,1)$. Supposons qu'à un essai précédent on ait estimé les utilités à 0.85 pour $(1,2)$ et à 0.95 pour $(2,1)$: si la transition considérée dans la séquence devait s'opérer quoiqu'il arrive, on devrait avoir $U^\pi(1,2) = -0.05 + U^\pi(2,1)$[14], et $U^\pi(1,2)$ vaudrait donc 0.90. Ainsi, il conviendrait d'augmenter l'utilité de $(1,2)$ qui semble un peu basse au regard de la transition $(1,2)$ vers $(2,1)$.

De manière concrète, quand il y a transition d'un état s à un état s', on met à jour la fonction d'utilité $U^\pi(s)$ de la manière suivante :

$$U^\pi(s) \leftarrow U^\pi(s) + \alpha\left(R(s) + \gamma U^\pi(s') - U^\pi(s)\right)$$

où α est le *taux d'apprentissage* : il rend compte de l'importance qu'on accorde à la valeur qu'est censée prendre $U^\pi(s)$ par rapport à $U^\pi(s')$. Remarquons que la mise à jour ne concerne que le successeur s' visité, alors que les conditions d'équilibre concernent tous les successeurs possibles.

Une implémentation de l'apprentissage par différence temporelle est présentée dans le Listing 5).

14. cf. Bellman, éq. 3.2, avec un facteur d'escompte $\gamma = 1$.

Listing 4 Fonction PASSIF-PDA-AGENT(`percept`)

Input : `percept` $= (s', r')$ {état courant s', récompense r'}
Output : a {action}
Statique :

- π {stratégie}
- X {pdm de paramètres T, R, γ}
- N_{sa} {fréquences des couples état-action, initialement vide}
- $N_{sas'}$ {fréquences des triplets état-action-état, initialement vide}
- s, a {états et action précédents}

if $\texttt{NOUVEAU}(s')$ **then**
  $U[s'] \leftarrow r'$
  $R[s] \leftarrow r'$
end if
if $s \neq null$ **then**
  $N_{sa}[s,a] \leftarrow N_{sa}[s,a] + 1$
  $N_{sas'}[s,a,s'] \leftarrow N_{sas'}[s,a,s'] + 1$
  for all t tel que $N_{sas'}[s,a,t] \neq \texttt{null}$ **do**
    $T[s,a,t] \leftarrow \frac{N_{sas'}[s,a,t]}{N_{sa}[s,a]}$
  end for
end if
$U \leftarrow \texttt{VALUE} - \texttt{DETERMINATION}(\pi, U, X)$
if $\texttt{TERMINAL}(s')$ **then**
  $(s,a) \leftarrow (null, null)$
else
  $(s,a) \leftarrow (s', \pi(s'))$
end if
return a

où `TERMINAL` est une fonction retournant un booléen **true** si l'état considéré est un état terminal, **false** sinon, et $\texttt{VALUE} - \texttt{DETERMINATION}$ détermine U grâce aux équations de Bellman.

3.4.3 Apprentissage par renforcement actif

Au contraire de l'agent passif, l'agent actif doit décider *au fur et à mesure* quelles actions effectuer - sa stratégie n'est pas fixée. L'agent devra apprendre un modèle complet avec des probabilités de résultat pour ses actions au lieu du modèle associé à une politique fixe. On pourra utiliser `PASSIF-PDA-AGENT` pour cela. L'agent dispose d'un ensemble d'actions possibles A. Les utilités qu'il doit estimer sont celles que définit la *stratégie optimale*, qui sont telles que :

$$U(s) = R(s) + \gamma \max_a \sum_{s'} T\left(s, a, s'\right) U(s')$$

On obtient de la sorte autant d'équations que d'états. Ce système d'équation peut se résoudre en utilisant un algorithme généralement appelé `VALUE-ITERATION` : après avoir initialisé les utilités à des valeurs arbitraires, on calcule la partie droite de l'équation et on met ainsi à jour l'utilité de chaque état, en répétant

Listing 5 Fonction PASSIF-TD-AGENT(`percept`)

Input : `percept` $= (s', r')$ {état courant s', récompense r'}
Output : a {action}
Statique :

- π {stratégie}
- U {table d'utilités, initialement vide}
- N_s {fréquences des états, initialement vide}
- s, a, r {état, action et récompense précédents}

if `NOUVEAU`(s') **then**
  $U[s'] \leftarrow r'$
end if
if $s \neq null$ **then**
  $N_s[s] \leftarrow N_s[s] + 1$
end if
$U[s] \leftarrow U[s] + \alpha N_s[s] \left(r + \gamma U[s'] - U[s]\right)$
if `TERMINAL`(s') **then**
  $(s, a, r) \leftarrow (null, null, null)$
else
  $(s, a, r) \leftarrow (s', \pi(s'), r')$
end if
return a

l'actualisation jusqu'à converger vers l'équilibre [15] :

$$U_{i+1}(s) = R(s) + \gamma \max_a \sum_{s'} T\left(s, a, s'\right) U_i(s')$$

On peut prouver qu'un équilibre est atteint presque sûrement en effectuant suffisamment d'itérations. Finalement, on dispose d'une fonction d'utilité optimale pour le modèle appris. Reste à savoir **quelle** action l'agent a intérêt à appliquer.
En effet, ce dont ne tient pas compte l'agent, c'est que les actions ne se résument pas à un gain de récompense, fonction du modèle déjà appris, mais contribuent également à l'apprentissage du *vrai* modèle. Si le modèle appris devient de plus en plus proche de celui-ci, l'agent recevra à l'avenir des récompenses plus importantes.

Ce dernier doit donc trouver un compromis entre l'*exploitation* qui lui permet de maximiser sa récompense - telles que les estimations d'utilités courantes la reflètent - et l'*exploration* qui pourrait maximiser son bien-être à long terme (c'est-à-dire obtenir une récompense totale supérieure à l'issue de la partie).

Q-learning

Au lieu d'apprendre les utilités, on se préoccupe cette fois-ci de la *valeur* d'une action. Soit $Q(a, s)$ la valeur d'une action a dans l'état s : les *Q-valeurs* représentent la qualité d'une action pour un état donné. Plus précisément, $Q(a, s)$ est la récompense immédiate obtenue en effectuant l'action a, à laquelle

15. De fait, on applique une méthode itérative de résolution d'équation de point fixe.

on ajoute la valeur obtenue en suivant la politique optimale par la suite. Les $Q(a, s)$ sont par conséquent liées aux utilités par :

$$U(s) = \max_a Q(a, s)$$

Les Q-fonctions possèdent une propriété importante : *les agents fondés sur une approche par différence temporelle qui apprennent les Q-fonctions n'ont pas besoin de modèle, ni pour apprendre, ni pour sélectionner les actions.* Écrivons l'équation de contrainte découlant de la définition de Q :

$$Q(a, s) = R(s) + \gamma \sum_{s'} T(s, a, s') \max_{a'} Q(a', s')$$

On peut toujours utiliser cette équation directement comme équation de mise à jour dans un processus d'itération qui calcule des Q-valeurs exactes étant donné un modèle estimé. Toutefois, cela nécessite d'apprendre aussi un modèle, du fait de l'utilisation de $T(s, a, s')$. En revanche, pour la méthode de différence temporelle, on écrit la relation :

$$Q(a, s) = Q(a, s) + \alpha \left(R(s) + \gamma \max_{a'} Q(a', s') - Q(a, s) \right)$$

Ce qui permet de mettre à jour $Q(a, s)$ chaque fois qu'une action est exécutée dans un état s menant à s'.

Listing 6 Fonction Q-LEARNING-AGENT(`percept`)

Input : `percept` $= (s', r')$ {état courant s', récompense r'}
Output : a {action}
Statique :
- Q {table de valeurs, indicée par couple état-action}
- N_{sa} {table de fréquences, indicée par couple état-action}
- s, a, r {état, action et récompense précédents}

if $s \neq null$ **then**
  $N_{sa}[s, a] \leftarrow N_{sa}[s, a] + 1$
  $Q(a, s) \leftarrow Q(a, s) + \alpha N_{sa}[s, a] \left(r + \gamma \max_{a'} Q[a', s'] - Q[a, s] \right)$
end if
if `TERMINAL`(s') **then**
  $(s, a, r) \leftarrow (null, null, null)$
else
  $(s, a, r) \leftarrow (s', \operatorname{argmax}_{a'} f(Q[a', s'], N_{sa}[a', s']), r')$
end if
return a

où $f(u, n)$ est la *fonction d'exploration* : elle détermine dans quelle mesure sacrifier la “gourmandise” (préférence pour des valeurs élevées de l'utilité u) à la “curiosité” (préférence pour les faibles valeurs de n - actions qui n'ont pas été essayées souvent).

Chapitre 4

Notre projet : spécificités et notions théorique nécessaires à sa bonne poursuite

4.1 Spécificités du projet et conséquences

4.1.1 Les besoins d'un jeu vidéo en termes d'intelligence artificielle

Notre projet vise à l'élaboration d'un jeu vidéo de type RPG (*Role-Playing Game*, ou encore jeu de rôle) dont les personnages non joueurs (PNJ [1]) seraient dotés d'une véritable intelligence artificielle, rendant leur comportement réaliste et le jeu plus intéressant pour le joueur.
Toutefois, ce type de jeu comporte des contraintes particulière, qu'il nous faut considérer :

- L'action se déroule en temps réel, d'où la nécessité d'une prise de décision rapide de la part des agents. *(Vitesse d'exécution)*
- Afin de simuler le comportement d'un personnage réel, les agents n'ont pas accès à la totalité de l'information du jeu - et ce d'autant plus que leur omniscience allongerait considérablement le temps de calcul nécessaire à la prise de décisions. *(Information incomplète)*
- Les situations auxquelles un personnage doit faire face sont extrêmement nombreuses (l'état d'un agent et les positions relatives des autres personnages sont déjà des paramètres possédant un grand nombre de valeurs possibles) *(Taille de l'espace d'états)*

En outre, nous souhaitons que le jeu puisse évoluer *en fonction du joueur*, c'est-à-dire que les PNJ adaptent leurs décisions à la façon de jouer de celui-ci. En d'autres termes, l'intelligence des PNJ ne doit pas être figée, mais continuer d'évoluer en apprenant de ses échecs et de ses succès face au joueur.

Enfin se pose la question du partage de connaissances : les PNJ partagent-ils la même intelligence artificielle, ou bien simplement le même algorithme, mettant ensuite leur stratégie à jour chacun de son côté ?

1. Nous emploierons indifféremment, par la suite, les termes *PNJ* et *agent*.

4.1.2 Premier pas vers le jeu : un prototype . . . non jouable

La première étape consiste à créer un environnement dans lequel n'évoluent que des PNJ, afin d'observer leur comportement, l'efficacité de l'intelligence artificielle utilisée, et éventuellement l'apparition de comportements émergents : ce prototype portera le nom d'*Highlander*.

Définition : Highlander

Nous commençons par un environnement simpliste, de petite taille et sans obstacles, dans lequel quelques PNJ sont « livrés à eux-mêmes ». L'environnement, appelé *carte*, est discret, et découpé en cases qui peuvent être occupées par un PNJ, un point de ravitaillement, ou vides. Les règles du jeu sont les suivantes :

- à chaque unité de temps (« tour »), les agents voient leur faim augmenter. Celle-ci est initialement à 0, et est incrémentée à chaque tour.
- lorsque la faim d'un agent arrive à 100, celui-ci meurt.
- des points de ravitaillement sont répartis sur la carte ; lorsqu'un agent passe à côté d'un de ces points, sa faim est réinitialisée à 0.
- lorsque deux agents se croisent, un combat s'engage inévitablement. Le vainqueur est désigné de manière aléatoire [2].
- au début de chaque partie, tous les agents sont de niveau 0. Celui-ci augmente de 1 à chaque meurtre perpétré.
- la partie se termine au bout d'un nombre de tours fixé, ou bien lorsqu'il ne reste qu'un survivant.

Caractéristiques du premier prototype

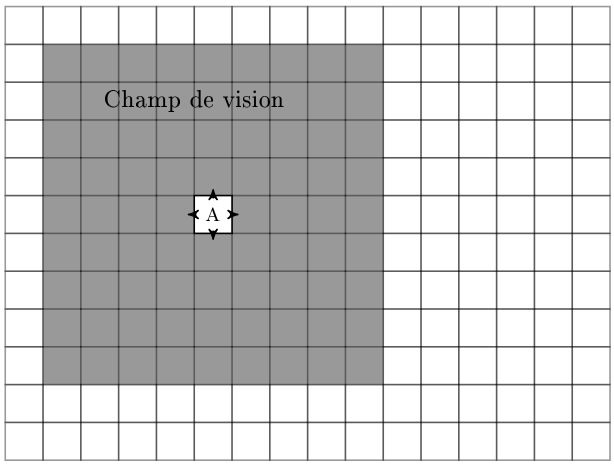

FIGURE 4.1 – Carte, agent et actions possibles

Un PNJ peut voir tout ce qui se trouve dans le carré de 9 cases de côté dont il est le centre. Il ne voit rien au-delà, mais peut se rappeler la position des points de ravitaillement.
À chaque tour, il a le choix entre 5 actions (voir Figure 4.1) :

2. Tirage de type *Roue de la fortune* : la probabilité de victoire d'un agent est proportionnelle à son niveau.

`UP` : aller vers le haut
`DOWN` : aller vers le bas
`RIGHT` : aller vers la droite
`LEFT` : aller vers la gauche
`NONE` : ne rien faire

La prise de décision de l'action à effectuer dépend de l'*état* dans lequel se trouve le PNJ. Cet état se compose de :

- la *direction* de l'ennemi le plus proche visible (*null* s'il n'y en a pas)
- la *distance* de l'ennemi le plus proche (5 si pas d'ennemi visible [3])
- le *niveau relatif* de l'ennemi le plus proche :
  0 si pas d'ennemi visible, ou si l'ennemi est de même niveau
  1 si l'ennemi est un peu plus fort
  2 s'il est bien plus fort
  -1 s'il est plus faible
  -2 s'il est bien plus faible
- l'*état de faim* H (0 si la faim est inférieure à 70, 1 sinon [4])
- la *direction* du point de nourriture le plus proche, visible ou mémorisé (*null* par défaut)
- la *distance* du point de nourriture le plus proche (5 s'il n'est pas visible)

La taille de la carte est de 100×100 cases. Nous y positionnons, aléatoirement, 9 points de nourriture fixes et 10 PNJ. Les parties durent 300 tours au maximum, et l'on fait participer les mêmes agents à de nouvelles parties jusqu'à obtenir des résultats satisfaisants.

4.2 Utilisations potentielles des réseaux de neurones

La richesse des réseaux de neurones et leur capacité à répondre à un problème de façon autonome leur ouvre un spectre d'applications très large, allant de la reconnaissance de caractères typographiés à la classification de données. Dans le cadre de notre de projet, nous avons pensé à deux applications, l'une participant directement de l'**intelligence des personnages** du jeu, et l'autre jouant un rôle d'**aide au traitement d'informations** devant être utilisées par une autre forme d'IA.

La première application concerne l'utilisation des réseaux de neurones dans le cadre du jeu lui-même, au sein de l'intelligence artificielle des personnages. En effet, les réseaux de neurones peuvent être utilisés comme outils de prise de décision adaptatifs : à partir d'un état, le réseau calcule la sortie qui représente un vecteur dans l'espace des décisions possibles. On peut ensuite envisager pour ce réseau un mode d'apprentissage supervisé, préalable au jeu, au cours duquel on soumet le réseau à des couples *(état, décision attendue)*. On peut aussi entraîner ce réseau en temps réel, en utilisant comme indice de qualité le résultat

3. Une distance de 5 correspond à l'éloignement d'un ennemi située « juste hors de portée », c'est-à-dire à la lisière du champ de vision.

4. $H = \mathbb{1}_{[\![0;70[\![}$

de la décision (par exemple, si la décision entraîne la mort du personnage, on applique une règle de Hebb [5] inhibitrice). Ce type d'utilisation des réseaux neuronaux permettrait de cette manière de fournir à l'intelligence artificielle une **composante évolutive de réponse à l'environnement** [BS04].

La seconde application représente davantage une aide à la mise en place de l'intelligence artificielle. En effet, comme l'espace des états d'un personnage est soit continu, soit trop grand, il est nécessaire de partitionner cet espace en zones d'états « équivalents » (i.e. amenant à la même décision) [6]. À cette fin, on peut utiliser la propriété des perceptrons de partitionnement de l'espace des entrées afin de regrouper les états par « paquets », ou pavés. On crée ainsi un *cluster* de l'espace des états d'un personnage, sans toutefois avoir à construire ce pavage de toute pièce - mais en le laissant se construire de façon optimale [Cha03].

Pour conclure cette partie, les réseaux de neurones - bien que peu utilisés à ce jour dans les jeux - présentent des avantages incontestables de par leur fonctionnement évolutif et autonome. Cependant, les tests concernant la mise en œuvre de ces potentialités sont encore à faire, afin d'obtenir un compromis optimal entre fiabilité de l'intelligence (et donc complexité du réseau) et capacités de calcul. En effet, n'oublions pas que l'environnement du jeu comporte des contraintes de gestion des temps de calcul et de la mémoire importantes [7], contraintes bien moins critiques dans les domaines d'application courants des réseaux neuronaux tels que le traitement d'image.

4.3 Pistes d'intégration des algorithmes génétiques

Nous n'avons pas, à ce jour, déterminé exactement quelle serait la meilleure façon d'intégrer une intelligence artificielle fondée sur des algorithmes génétiques dans le scénario *Highlander*. Toutefois, différentes possibilités ont été évoquées, notamment :

- appliquer ces algorithmes non aux agents eux-mêmes, mais à leur stratégie - chaque agent disposant initialement d'une stratégie donnée (fixée en grande partie aléatoirement), n'évoluant pas en cours de partie.
- attribuer à chaque agent un ensemble de *caractéristiques* ("curiosité", "lâcheté", "gourmandise" ...) qui détermine son comportement. C'est sur ces caractéristiques, durant ou à la fin de chaque partie, que porteraient les évolutions et l'apprentissage.

D'autres options sont bien entendu susceptibles de se révéler plus pertinentes ; cependant, nous serons plus à même d'en juger lorsque nous disposerons des premières observations sur le comportement des deux autres types d'IA considérés pour *Highlander*(à savoir les réseaux neuronaux, section 4.2, et l'apprentissage par renforcement, section 4.4).

5. cf. section 3.2.2, p.31
6. On parle de *clustering*, ou classification des états.
7. cf. section 4.1.1, p.44

4.4 Apprentissage par renforcement

Comme on l'a vu précédemment (section 3.4), l'apprentissage par renforcement concerne l'apprentissage par un agent autonome d'une *politique optimale*, c'est-à-dire de l'action la mieux adaptée à chaque situation envisageable pour le système décisionnel considéré. La structure de son environnement étant généralement supposée inconnue, l'agent doit apprendre à partir de ses interactions avec le monde. En particulier, aucun professeur ne lui dit quelle action est la meilleure à prendre dans une situation donnée, et seul un *signal de renforcement* assez peu informatif (un scalaire) l'informe de temps en temps de sa performance liée à ses décisions passées.

Classiquement, l'apprentissage par renforcement est basé sur une fonction d'utilité. Divers algorithmes et structures de représentations de l'environnement ont été proposés pour apprendre cette fonction d'utilité dans le cadre formel des processus décisionnels markoviens (PDM).

Dans ces approches, l'agent cherche à apprendre l'utilité de chaque état ou de chaque paire *(état, action)* (c'est notre cas). Il sélectionne alors l'action associée à l'utilité maximale. Si la fonction d'utilité estimée est exacte, cette approche conduit à la politique optimale sous des conditions très générales. Cependant, pour les plupart des problèmes du monde réel, il est impossible de représenter les fonctions d'utilité exactement, par exemple comme tables de valeur - l'agent doit alors chercher une bonne approximation de la fonction d'utilité au sein d'une classe restreinte de fonctions (par exemple sous la forme d'un réseau neuronal ou d'une classe de *fonctions noyau*[8]). La difficulté est de trouver une représentation compacte assurant la convergence des méthodes d'apprentissage pour les PDM. C'est pourquoi de nouvelles approches d'apprentissage plus directes de la politique sont aussi explorées.

Si des réponses très complètes ont pu être apportées par ailleurs ([AC02], par exemple), nous nous contenterons dans ce rapport de définir le cadre conceptuel.

Stimuli pour *Highlander*

Pour mieux comprendre de quoi il s'agit concrètement, voici des exemples de *stimuli* (récompenses) possibles dans le cas de notre scénario « Highlander ». À noter que plus il y a de stimuli (pertinents), et plus ils sont proches de l'action qu'ils sanctionnent, plus l'apprentissage sera rapide.

+ après un combat gagné
+ après avoir mangé (signal inversement proportionnel à l'état de faim antérieur)
+ lorsque le niveau de faim est bas
+ en cas de survie à la fin d'une partie
- lors de la mort de l'agent
- lorsque le niveau de faim est élevé

8. Une fonction noyau *(kernel function)* est une fonction de pondération utilisée en statistique, dans les techniques d'estimation non-paramétrique. Il s'agit d'une fonction positive, paire, intégrable sur $\mathbb{R}$, et d'intégrale égale à 1.

Connaissance de l'environnement dans *Highlander*

Comme nous l'avons mentionné dans la partie précédente (section 4.1), la connaissance de l'environnement n'est que partielle (champ de vision limité à une boule[9] centrée sur le personnage). Ainsi, plusieurs observations identiques peuvent correspondre à différents états[10]. Cela complique le problème - une solution peut être de garder une mémoire des actions passées, ou de prendre en compte l'incertitude sur les états (en considérant un nouvel espace qui tient compte des probabilités d'une observation, à un instant et un état donné).

Si nous observons des problèmes de convergence des algorithmes, une méthode serait de commencer par de l'apprentissage passif (ie, munir les agents d'une stratégie préexistante, voir section 3.4 p.37), et de ne tâcher d'implémenter de l'apprentissage actif (où ils définissent eux-mêmes leur stratégie) que dans un second temps.

Dilemme exploration contre exploitation

Notre agent ne connait pas son environnement, et cherche à construire une mesure de gain cumulée. Pour ce faire, il doit naturellement commencer à explorer son univers pour en découvrir les potentialités - mais très vite il se retrouve confronté à un dilemme : ayant découvert que certains comportements semblent plus profitables que d'autres, doit-il chercher à les reproduire au maximum afin d'augmenter le gain recherché, au risque de passer à côté d'opportunités non encore identifiées, ou doit-il continuer son exploration, quitte à éventuellement perdre du temps et ne pas réaliser un bon gain cumulé ?

Ce problème est classique lorsqu'un agent doit prendre des décisions « en ligne », et est modélisé sous la forme d'un *bandit à deux bras* (voir ci-dessous). Différentes solutions existent, qui peuvent être divisées en deux catégories : les méthodes dites *non dirigées*[11], qui reposent sur les évaluations des actions, et les méthodes *dirigées* (méthode basée sur la « récence »[12], estimation d'incertitude), qui utilisent en plus des heuristiques exploitant des informations acquises lors de l'apprentissage.

Bandit à deux bras (ou *bandit manchot*)

Le principe de ce problème, inspiré des machines que l'on pouvait trouver dans les casinos, est le suivant : un joueur dispose de m jetons avec lesquels il peut jouer avec une machine à sous à deux bras. Pour chaque jeton inséré dans la fente, le joueur peut tirer sur l'un ou l'autre des bras. Il reçoit alors un certain nombre de pièces correspondant à son gain. Les bras sont Notés A_1 et A_2, ils ont une espérance de gain respective de μ_1 et μ_2, avec une variance respective de σ_1^2 et σ_2^2. Cela signifie que le gain associé à chaque bras est aléatoire, avec une certaine moyenne et un certain écart-type stationnaires. Les tirages aléatoires sont supposés indépendants. Le joueur ne connaît ni les moyennes ni les variances associées à chaque bras et doit donc les estimer en cours de jeu. Il ne sait donc

9. *Boule* au sens topologique du terme : de par la distance choisie, les boules en question sont ... carrées.
10. On parle dans ce cas d'*aliasing*, ou situations d'alias perceptuel.
11. Meilleure action, ε-greedy, softmax ...
12. C'est-à-dire favorisant les actions n'ayant pas été choisies depuis longtemps.

pas quel est le bras dont l'espérance de gain est la meilleure, et doit essayer de maximiser son gain avec ses m jetons.

Quelle doit alors être sa stratégie de tirage des bras, étant donnée son estimation courante des moyenne et variance de chaque bras ?

Une stratégie extrême est de tirer une fois chaque bras, de noter celui qui a donné le meilleur résultat, puis de jouer désormais systématiquement celui-ci. Cela correspond à une stratégie d'exploitation pure : ne plus explorer dès que l'on possède des éléments d'information minimaux. Bien sûr, il existe un risque que les deux tirages initiaux n'aient pas révélé le meilleur bras à cause de la variance des résultats et qu'en conséquence ce soit le plus mauvais bras qui ait été tiré systématiquement à partir de là. Plus généralement, on appelle *stratégie d'exploitation* toute stratégie qui choisit l'action dont l'estimation courante de gain est la plus élevée.

La stratégie extrême inverse consiste à tirer $\lfloor \frac{m-1}{2} \rfloor$ fois sur le bras gauche, et autant sur le bras droit, puis à tirer le ou les deux derniers coups sur le bras dont la moyenne observée est la meilleure. Cela correspond à une *exploration pure*, dans quelle on alloue quasiment toutes les décisions à l'exploration de l'environnement avant de choisir la décision ultime. Celles-ci est alors prise avec une connaissance aussi grande que possible, mais au prix de n'avoir pas cherché à optimiser le gain durant la phase d'exploration.

On sent que la stratégie optimale doit se situer entre ces deux types de politiques, et qu'elle correspond à la résolution d'un **compromis entre exploitation et exploration**.

4.5 Agents et protocole

Dans le cadre d'*Highlander*, nous nous focaliserons sur des **agents à utilité avec apprentissage** : de la sorte, nous serons en mesure d'implémenter les différentes méthodes d'intelligence artificielles présentées dans ce document, en particulier l'apprentissage par renforcement (section 3.4), les algorithmes génétiques (section 3.3) et les réseaux neuronaux (section 3.2).

De fait, nous tâcherons de mettre en présence une population initiale équitablement répartie entre trois types d'agents[13], chacun disposant en interne d'un paradigme d'intelligence différent ; le but sera alors d'*alterner des phases d'apprentissage (entraînement) inter- et intra-populations*(voir Figure 4.2). De la sorte, nous sélectionnerons les éléments les plus « prometteurs » de chaque population, puis les confronterons à d'autres agents (ceux des deux autres populations), dont l'apprentissage se déroule de manière différente (vitesse de convergence, type d'exploration des actions et possiblités ...).

Après chaque séquence, comportant un nombre n à définir de phases inter/intra (sur la figure, $n = 1$), nous relèverons les résultats obtenus, afin de

13. Désignés sur le schéma par *NN* (Neural Network), *GA* (Genetic Algorithm) et *RL* (Reinforcement Learning). *Highlander*, enfin, sera abrégé en *HL*.

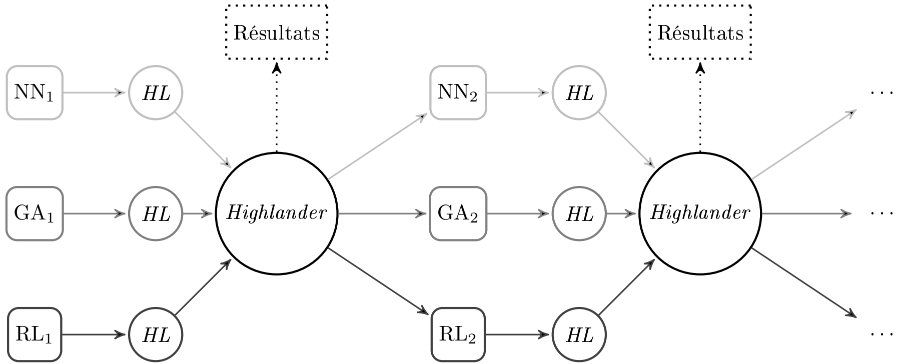

FIGURE 4.2 – Protocole adopté

dresser une sorte de **comparatif multicritère** des 3 types d'intelligence artificielle appliqués à notre scénario *Highlander*. Si nous en avons le temps, nous pourrons par la suite étendre ce protocole à d'autres scénarios, comportant des règles et environnements différents.

De la sorte, nous serons en mesure d'apporter des éléments de réponse à plusieurs des interrogations qui sous-tendent notre projet, à savoir :

- *L'intégration d'intelligence artificielle dans des productions vidéoludiques est-elle réalisable en pratique ?*
  De fait, la vitesse de convergence des différents algorithmes et la rapidité d'apprentissage des agents est ici critique ; si, au terme d'un entraînement équivalent à une dizaine d'heures de « jeu », les PNJ font toujours preuve d'un comportement simpliste et d'une intelligence rudimentaire, leur présence dans un jeu vidéo grand public risque de se révéler tout sauf concluante - de même si leur vitesse d'apprentissage est non perceptible sur la durée de vie d'un jeu.
- *Quelle sorte d'IA choisir, dans quelles circonstances ?*
  Nous espérons également pouvoir, munis de nos résultats, émettre des recommandations quant à l'utilisation de tel ou tel type d'IA, dans l'application à un genre de problème particulier, en fonction du critère à privilégier (rapidité d'apprentissage lorsque confrontée aux actions du joueur, comportements émergents, recherche de résultats satisfaisants à plus ou moins brève échéance ...).

Chapitre 5

Conclusion

« La faculté de citer est un substitut commode à l'intelligence. »
(William-Somerset Maugham)

Comme cela a déjà été rappelé, l'industrie du jeu vidéo est un marché en plein essor. Si certaines pistes d'amélioration (types de jeu, scénarios, graphismes, etc.) ont déjà été explorées, sans toutefois avoir atteint leurs limites, il semble que l'intelligence artificielle soit encore peu répandue dans les jeux vidéo. Pourtant, des études de plus en plus nombreuses tendent à confirmer qu'elle pourrait être intégrée dans des jeux de divers types. C'est dans cette optique que nous nous sommes tout d'abord intéressés à sa définition et aux diverses formes qu'elle peut prendre.

L'objectif de notre projet repose sur l'élaboration d'un prototype de jeu utilisant l'intelligence artificielle, mais pas de n'importe quel type : en effet, nous avons préféré nous concentrer sur l'utilisation d'une IA évolutive, au détriment de l'IA scriptée plus communément utilisée dans les jeux vidéo. Ce choix nous a ainsi conduit à envisager différents types d'intelligence artificielle, notamment les réseaux de neurones, les algorithmes génétiques et l'apprentissage par renforcement, dont on sait que l'implémentation dans des jeux vidéo est possible et donne même des résultats intéressants.

Certaines de nos recherches nous ont même amenés à envisager le couplage de ces différents types d'IA, l'un agissant sur la structure d'un autre. Par exemple, une équipe de chercheurs a récemment réussi à implémenter des algorithmes génétiques qui agissent directement sur la structure d'un réseau neuronal [Sta05]. Cette mise en relation de deux IA différentes a donné de très bons résultats, et représente une perspective de recherche intéressante qu'il faudrait peut-être songer à approfondir.

Bibliographie

[AC02] Yves Kodratoff Antoine Cornuéjols, Laurent Miclet. *Apprentissage Artificiel : Concepts et algorithmes*. Eyrolles, 1st edition, 2002. fr. 13, 18, 48

[BS04] David M. Bourg and Glenn Seemann. *AI for Game Developers*. O'Reilly Media, Inc., 2004. 47

[Cha03] Alex J. Champandard. *AI Game Development*. New Riders Games, 2003. 47

[DSW09] Thomas Degris, Olivier Sigaud, and Pierre-Henri Wuillemin. Apprentissage par renforcement factorisé pour le comportement de personnages non joueurs. *Revue d'Intelligence Artificielle*, 23(2-3) :221–251, 2009. 21

[FS91] James A. Freeman and David M. Skapura. *Neural networks : algorithms, applications, and programming techniques*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1991. 32

[Gel07] Sylvain Gelly. *A Contribution to Reinforcement Learning ; Application to Computer-Go*. PhD thesis, Université Paris-Sud, 2007. 15

[LR85] T. L. Lai and Herbert Robbins. Asymptotically efficient adaptive allocation rules. *Adv. in Appl. Math.*, 6(1) :4–22, 1985. 14

[ODD03] Ales Oulladji, Latifa, Jean-Antoine Désidéri, and Alain Dervieux. Optimisation aérodynamique par algorithmes génétiques hybrides : application à la réduction d'un critère de bang sonique. 0 RR-4884, INRIA, Juillet 2003. 37

[Par04] Marc Parizeau. Réseaux de neurones (notes de cours). *Cours de l'Université Laval*, Automne 2004. 30, 33

[RF07] Denis Robillard and Cyril Fonlupt. Cours de programmation génétique. Technical report, Laboratoire d'informatique du Littoral, Univ. Littoral-Côte d'Opale, Calais, 2007. 37

[RN03] Stuart Russell and Peter Norvig. *Artificial Intelligence : A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition, 2003. 3, 25

[Saf08] Abdallah Saffidine. Utilisation d'UCT au Hex. pages 6,7, 2008. 15

[Sam59] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3) :210–229, 1959. 17

[Sta05] Kenneth O. Stanley. Evolving neural network agents in the NERO video game. In *In Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games*, pages 182–189, 2005. 52

[Tes95] Gerald Tesauro. Temporal difference learning and TD-gammon. *Commun. ACM*, 38 :58–68, March 1995. 17

[Tor08] Fabien Torre. L'intelligence artificielle et les jeux. en ligne, Août 2008. 9

[VY01] Thomas Vallee and Murat Yildizoglu. Présentation des algorithmes génétiques et de leurs applications en économie. Technical report, Université de Nantes, LEA-CIL ; Université Montesquieu Bordeaux IV, Septembre 2001. 37