

Université catholique de Louvain
Faculté des Sciences Appliquées
Département d'Ingénierie Informatique

Application du “Reinforcement
Learning” à un jeu de Markov de type
évasion-poursuite

Mémoire présenté en vue
de l'obtention du grade
d'Ingénieur Civil
en Informatique
par Lionel DRICOT

Promoteur : Professeur Marco SAERENS
Lecteurs : Youssef ACHBANY & François FOUSS

Louvain-la-Neuve
Août 2006

Résumé

Ce mémoire étudie le problème évaison-poursuite dans le cas d'un "Markov Decision Process" et dans le cas d'un jeu de Markov. Le problème est ici symbolisé par un ou plusieurs chats cherchant à attraper une souris dans un espace discret torique. La méthode de "Q-Learning" pour résoudre ce problème de "Reinforcement Learning" est étudiée. Il est aussi nécessaire de prévoir une coopération entre les chats afin d'atteindre l'objectif. Une méthode de coopération dite "agent et sous-agents" est développée et testée. L'implémentation en Python d'un framework généraliste typique pour ce genre de problème est ensuite décrite en détails avec les résultats obtenus pour la méthode de coopération proposée.

Table des matières

Introduction	5
Remerciements	6
1 Les bases du Reinforcement Learning	8
1.1 Agent et Environnement	8
1.2 Rewards - Bonus - Malus	10
1.2.1 Introduction au return - bonus - malus	10
1.2.2 Le Return attendu	11
1.3 Valeur d'un état	12
1.4 Dilemme Exploitation - Exploration	13
1.5 En résumé	14
2 Les jeux de Markov	16
2.1 La propriété de Markov	16
2.1.1 L'approximation markovienne	18
2.2 Les Markov Decision Processes	19
2.2.1 Calcul des valeurs dans un Markov Decision Process	19
2.2.2 Recherche de la valeur optimale	20
2.2.3 Méthode de résolution	21
2.3 Les jeux de Markov	22
2.4 En résumé	23
3 Le problème du chat et de la souris	25
3.1 Définition du problème	25
3.2 Espace du problème	26
3.2.1 Calcul de la distance	26
3.3 L'agent chat	28
3.3.1 Coopération entre les chats	29
3.3.2 Agents et sous-agents	29
3.3.3 Apprentissage et reward	30
3.4 L'agent souris	32

3.4.1	L'algorithme best-escape	32
3.5	Obstacles	33
3.6	Récapitulation : agents et environnement	34
3.7	Analyse du problème	34
3.7.1	2 chats et une souris	34
3.7.2	3 chats et une souris	36
3.8	Dilemme exploration-exploitation	37
3.8.1	Exploration stochastique fixe	37
3.8.2	Exploration stochastique décroissante	38
3.8.3	Entropie et degré d'exploration	39
3.9	En résumé	41
4	Implémentation d'un framework de Reinforcement Learning	42
4.1	L'environnement	42
4.1.1	La distance	43
4.1.2	Identification d'un état	43
4.2	L'agent	45
4.2.1	L'objet agent	45
4.2.2	L'objet subagent	47
4.3	Le chat	48
4.3.1	Actions possibles	48
4.3.2	Reward	49
4.3.3	Q-Learning	50
4.3.4	Exploration/Exploitation	51
4.4	La souris	52
4.5	Déroulement du programme	53
4.6	Visualisation	54
4.7	En résumé	56
5	Simulations et expériences	57
5.1	Recherche classique dans l'arbre des solutions sans coopération	57
5.1.1	Le problème du parallélisme et la symétrie du problème	57
5.2	Mode opératoire	58
5.3	Expérience 3 chats et une souris	59
5.3.1	Situations de test	60
5.3.2	Calcul du paramètre <i>explore_decrease</i>	60
5.3.3	Influence de l'apprentissage	61
5.3.4	Influence du coefficient d'exploration	65
5.3.5	Analyse de l'expérience	66
5.4	Expérience 2 chats et une souris fatiguée	66
5.4.1	Conditions de l'expérience	66

5.4.2	Influence de l'apprentissage	67
5.5	Conclusions de l'expérimentation	68
Conclusion et perspectives		70
	Perspectives	71
Bibliographie		72
Annexe A : Utilisation du programme		74
	Lancement d'un run	74
	Implémenter un agent personnalisé	75
	Affichage personnalisé	76
Annexe B : Codes sources		78
	main.py	79
	values.py	83
	objets.py	85
	ml.py	92
	ml_envir.py	96
	display.py	101
	jeu1.py	103

Table des figures

3.1	Exemple du problème : deux chats (cercles noirs) chassant une souris (cercle blanc) dans un espace 10x10	26
3.2	Illustration de deux mouvements possibles dans un espace torique	27
3.3	Mouvements possibles d'un sous-agent	28
3.4	Les 6 façons pour 2 chats d'encercler une souris	35
4.1	Deux états rigoureusement identiques par translation	44
4.2	Implémentation de l'objet <i>objet</i> pour un obstacle	47
4.3	Gestion d'un espace torique dans l'implémentation du mouvement	48
4.4	Affichage de la matrice environnement en console par la méthode <code>display</code>	55
4.5	Affichage de l'environnement via la librairie LiveWires	55
5.1	Illustration du problème du parallélisme	58
5.2	Les deux situations initiales testées avec 3 chats/1 souris	60
5.3	Influence de l'apprentissage dans la situation 1	62
5.4	Influence de l'apprentissage dans la situation 2	62
5.5	Répartition des runs en fonction du nombre d'actions pour atteindre la souris. La situation 1 est en gris foncé, la situation 2 en pointillés.	64
5.6	Influence du paramètre <i>explore_decrease</i> dans la situation 2	65
5.7	La troisième situation de nos expériences : 2 chats et une souris	67
5.8	Influence de l'apprentissage dans la situation 3	68

Introduction

L'ordinateur est un outil extrêmement puissant qui permet de calculer en une fraction de seconde des dizaines d'opérations avec une précision extrême. Cependant, depuis les débuts de l'informatique, effectuer un calcul nécessite un programmeur capable de donner à la machine toutes les données du problème dans un langage précis. Il s'agit donc de réaliser un modèle du problème avant de laisser le soin à l'ordinateur de s'occuper de la partie calculatoire. La conception de ce modèle a toujours été du domaine de l'intelligence humaine.

Le Reinforcement Learning est une méthode moderne d'Intelligence Artificielle qui tente de combler cette lacune. Grâce au Reinforcement Learning, un ordinateur est capable d'explorer un environnement afin d'en déduire son propre modèle, modèle qui lui permettra de trouver la solution à un problème donné dans l'environnement.

Ce mémoire se concentre sur un problème concret qui, quoique simple en apparence, recèle une complexité théorique et expérimentale passionnante : le problème de poursuite-évasion. Notre ordinateur est donc ici un poursuivant (un chat) à la recherche d'un évadé (une souris). Notre chat va donc devoir, au fil de ses expériences, apprendre le comportement de la souris afin d'en déduire ses futurs mouvements et donc finir par l'attraper.

Dans le chapitre 1, nous introduisons les bases théoriques du Reinforcement Learning : agent, environnement, reward, exploration, exploitation. À ce stade, nous restons très généraliste et nous contentons de survoler le sujet.

Le chapitre 2 introduit le thème plus complexe de Markov : propriété de Markov, Markov Decision Process, Markov Games. Notre problème consiste en effet en un jeu de Markov de la plus pure espèce. Des méthodes de résolution sont discutées et nous nous attarderons sur la méthode du Q-Learning.

À la lumière des deux chapitres précédents, le chapitre 3 analyse en profondeur notre problème chat-souris. Nous expliquons les concepts fondamentaux à définir dans ce type de problème (distance, agent, sous-agent) et discutons de la coopération possible entre plusieurs chats. Nous introduisons une méthode de coopération un peu particulière qui consiste à diviser un seul agent en sous-agents dans le cadre du problème. Nous nous rendons aussi compte que sous des apparences simplistes, le problème peut se révéler étonnamment complexe.

Afin de mener à bien les expériences, ce mémoire a nécessité l'implémentation d'un framework complet de Reinforcement Learning en Python. La manière dont celui-ci a été réalisé est présenté au chapitre 4. L'implémentation et les choix techniques réalisés dans ce chapitre se basent principalement sur les résultats de [Littman 1994].

Enfin, le chapitre 5 présente quelques résultats obtenus avec le sus-dit framework.

Ce mémoire comporte donc deux objectifs majeurs :

- Le premier, technique, consiste en l'implémentation d'un framework en Python permettant de traiter de manière assez générale les problèmes de Reinforcement Learning.
- Le second, plus expérimental, cherche à déterminer si une coopération entre les agents va bel et bien émerger de la méthode agent/sous-agents proposée pour implémenter cette coopération.

Nous avons donc, dans ce mémoire, deux aspects très différents, l'un purement technique et le second très théorique. Ces deux aspects sont cependant étroitement imbriqués et c'est certainement l'une des particularités de ce mémoire.

Remerciements

Je tiens tout d'abord à remercier tout spécialement mon promoteur, le Professeur Marco Saerens, pour sa disponibilité, sa patience, ses conseils et ses idées.

Un grand merci à Bertrand et à tout le 127.1, de Manille à Cranfield en passant par la Boucle des Métiers, pour le soutien moral, ainsi qu'à mes pa-

rents, pour la relecture et le soutien logistique tout au long de l'année. Sans eux, il ne m'aurait pas été possible de mener à bien ces années d'étude.

Je tiens aussi à adresser un remerciement particulier aux Professeurs Jan Govaerts et Joseph Lemaire qui, sans être impliqués dans mon mémoire, m'ont apporté ce que je considère, tant du point de vue humain que scientifique, des briques particulièrement importantes de ma formation d'ingénieur.

Chapitre 1

Les bases du Reinforcement Learning

Lorsque nous jouons à un jeu, la plupart du temps nous prenons d'abord le temps de lire les règles. Une fois les règles comprises et assimilées, nous pouvons établir un plan d'action qui nous permettra d'atteindre l'objectif du jeu (tel que défini par les règles) tout en évitant les obstacles (tels que définis par les règles). Le monde du jeu est parfaitement connu et délimité et la difficulté réside généralement à réaliser une exploitation des règles optimale ou, en tout cas, meilleure que son adversaire.

Malheureusement, dans la vie réelle comme dans beaucoup de problèmes courants, il n'existe pas d'énoncé des règles. Comment réagir dans un environnement dont on ne connaît pas les règles et donc même pas l'objectif à atteindre ?

C'est pour résoudre ce genre de problème que nous allons utiliser le Reinforcement Learning, méthode par laquelle nous apprendrons les règles du monde qui nous entoure tout en les exploitant.

1.1 Agent et Environnement

Dans bon nombre de problèmes, nous souhaitons faire atteindre un but à un élément donné, appelé agent, sans connaître les règles pour atteindre le but en question. Le Reinforcement Learning est une méthode directe pour apprendre dynamiquement ces règles en interagissant avec l'environnement. Par environnement, nous entendons tout ce qui est extérieur à l'agent concerné [Sutton & Barto, 1998]. Dans des situations où plusieurs agents interagissent,

pour un agent donné l'environnement comprend donc les autres agents.

L'interaction entre l'agent et son environnement est permanente. L'agent choisit d'effectuer une "action" et l'environnement se modifie en fonction de l'action. Notre agent est donc confronté à un nouveau choix d'actions.

Remarquons que l'agent n'a pas nécessairement la perception complète de son environnement, cette restriction imposant une difficulté supplémentaire à la prise de décision. La perception peut aussi être biaisée et non-exacte, par exemple parce que les senseurs ont une marge d'erreur. Dans ce cas, il faudra recourir à la probabilité que la situation soit X en fonction d'une observation Y [Isler et al, 2004].

Nous allons tout d'abord considérer un problème dans un temps et un espace discret où la perception de l'agent est à la fois complète et exacte.

À chaque étape de temps $t = 0, 1, 2, 3, \dots$ l'agent reçoit une représentation de l'état de l'environnement, $s_t \in S$, où S est l'ensemble de tous les états possibles.

En fonction de s_t , l'agent choisit l'action $a_t \in A(s_t)$ où $A(s_t)$ est l'ensemble des actions possibles dans la situation s_t .

A chaque étape t , l'agent doit donc réaliser un mapping π_t où $\pi_t(s, a)$ est la probabilité d'effectuer l'action a dans la situation s au temps t . Ce mapping est appelé la "policy" de l'agent [Sutton & Barto, 1998].

Il est évident que si cette policy est fixée, nous nous retrouvons dans le cas classique de règles apprises à fortiori et non pas dans une situation d'apprentissage dynamique. Le but du Reinforcement Learning est donc de spécifier de quelle manière modifier la policy en fonction de l'expérience.

Ainsi, si l'agent est un aventurier perdu dans d'obscurs couloirs (environnement), l'on apprendra très vite que d'une situation où l'on est au bord d'une fenêtre (situation s), sauter par la fenêtre (action a) peut n'être pas idéal (= KO).

Le Reinforcement Learning modifiera donc la policy pour que la prochaine fois que notre aventurier se trouve au bord d'une fenêtre, la probabilité qu'il enjambe le rebord soit très faible.

1.2 Rewards - Bonus - Malus

1.2.1 Introduction au return - bonus - malus

Ainsi que le lecteur l'aura remarqué, nous n'avons pas spécifié de quelle manière le Reinforcement Learning peut tirer profit de l'expérience, expérience que nous avons symbolisée par "KO" dans l'exemple de notre aventurier.

Une particularité propre au Reinforcement Learning est l'utilisation d'un reward, bonus ou malus, après chaque action. Le reward est un nombre entier envoyé par l'environnement à l'agent à chaque étape de temps t .

Le but de l'agent sera de maximiser la somme totale des rewards reçus jusqu'à l'objectif final symbolisé par un reward de $+\infty$. Cette somme est appelée le "Return". Il est important de noter que l'agent ne cherche pas à maximiser le reward immédiat de la prochaine action mais bien le reward total, dans la durée. Cela peut conduire l'agent à effectuer des actions donnant lieu à un malus (sauter par la fenêtre) si c'est la seule manière d'atteindre l'objectif (par exemple si notre aventurier doit sortir du donjon).

Cette approche, à priori simple, se révèle relativement efficace et souple à l'utilisation. Ainsi, si l'on souhaite trouver la solution la plus courte, il suffit, par exemple, d'imposer un malus de -1 à chaque étape temporelle.

Notons qu'il est important de distinguer l'objectif de la manière de l'atteindre. Il est de la tâche du programmeur d'utiliser son intuition afin de ne pas donner des boni à la légère sur ce qui n'est pas un but en soi mais une manière d'atteindre le but. Typiquement, c'est une mauvaise idée de donner un bonus à un programme d'échec pour chaque pièce prise à l'adversaire. En effet, le programme peut alors s'évertuer à prendre toutes les pièces possibles plutôt que de tenter de mettre le roi adverse en position d'échec.

Remarquons que, contrairement à notre intuition humaine, des rewards positifs ne sont pas toujours nécessaires. Si uniquement des mali sont dispensés, notre agent essaiera de minimiser ces mali, le plus éclatant succès étant alors un reward total nul. Notre agent n'est pas une otarie à laquelle on donne des poissons et il n'est pas toujours indispensable de prévoir un bonus pour l'objectif.

1.2.2 Le Return attendu

Il est évident que le Return ne peut être calculé de manière exacte. Après tout, si c'était le cas, il n'y aurait pas de raisons de se casser la tête avec le Reinforcement Learning.

Plaçons notre agent au temps t . Quelle que soit la position actuelle et les actions effectuées, nous pouvons considérer que pour maximiser le reward total, il faut maximiser le total des rewards futurs. En effet, les rewards déjà engrangés ne peuvent de toute façon plus être modifiés.

Nous pouvons alors définir le Return attendu au temps t comme la somme des rewards espérés aux temps $t + 1$, $t + 2$, ... jusqu'au temps final symbolisé T .

$$R_t = r_{t+1} + r_{t+2} + \dots + r_T \quad (1.1)$$

Cependant, il apparaît vite que, en dehors de certaines règles de rewards particulièrement strictes, la solution idéale pour notre agent consistera à attendre un temps infini avant d'atteindre son objectif, engrangeant quelques boni au passage.

Pour éviter ce problème, [Sutton & Barto, 1998] introduit une définition légèrement modifiée du Return attendu, à savoir le Return attendu amorti :

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \quad (1.2)$$

Où le paramètre γ est le facteur d'amortissement et $0 \leq \gamma \leq 1$. Ce facteur d'amortissement permet d'introduire à notre agent la notion du "Un tiens vaut mieux que deux tu l'auras", notion selon laquelle plus un reward est lointain dans le temps, moins il doit être considéré.

Notons les cas particuliers où $\gamma = 0$, cas dans lequel notre agent est myope et ne cherche qu'à satisfaire un besoin immédiat, et $\gamma = 1$, cas dans lequel nous retombons dans la situation précédente.

La pondération intelligente des boni et mali en fonction du problème étudié ne peut, à l'heure actuelle, être automatisée et requiert l'intuition et l'expérience d'un humain.

1.3 Valeur d'un état

Avec le Return attendu, nous avons donc défini une manière de savoir si un état est bon ou pas. Au mieux sera le return attendu, au mieux sera l'état considéré.

Voyons à présent comment calculer ce return attendu et surtout, ce qui est le coeur même de notre problème, comment découvrir l'état optimal.

Rappelons qu'une policy π est un mapping des états $s \in S$ et actions $a \in A(s)$ vers la probabilité $\pi(s, a)$ d'effectuer une action a lorsqu'on est dans l'état s . Lorsque nous nous trouvons à l'état s en suivant la policy π , on note la valeur de s comme étant $V^\pi(s)$, le return attendu en partant de s et en suivant la policy π jusqu'à la résolution du problème.

Notons $E_\pi\{\}$ la valeur attendue (*Expected value*) sachant que l'on poursuit la policy π . Nous pouvons donc définir $V^\pi(s)$ comme étant :

$$V^\pi(s, p) = E_\pi\{R_t | s_t = s; s_{t-1} = s'; \dots; s_0 = s''\} \quad (1.3)$$

Où p représente le chemin pour arriver à l'état s :

$$p = s_{t-1} = s'; \dots; s_0 = s'' \quad (1.4)$$

L'équation 1.3 peut être explicitée suivant l'équation 1.2 en :

$$V^\pi(s, p) = E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s; s_{t-1} = s'; \dots; s_0 = s''\right\} \quad (1.5)$$

La fonction V^π est appelée la fonction valeur-état pour la policy π .

De la même manière, nous pouvons définir la valeur d'une action a effectuée dans un état s sous la policy π . Notons cette fonction $Q^\pi(s, a)$. Elle signifie qu'à l'état s nous avons choisi l'action a et ensuite suivit la policy π .

$$Q^\pi(s, a, p) = E_\pi\{R_t | s_t = s; a_t = a; s_{t-1} = s'; \dots; s_0 = s''\} \quad (1.6)$$

et donc, en toute logique selon l'équation 1.5, on trouve que

$$Q^\pi(s, a, p) = E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s; a_t = a; s_{t-1} = s'; \dots; s_0 = s''\right\} \quad (1.7)$$

Selon [Sutton & Barto, 1998], Q^π est appelé la fonction valeur-action pour la policy π .

Le coeur de notre problème se résume donc à estimer correctement V^π et Q^π .

Supposons que nous disposions des mesures d'une infinité d'observations. Il apparaît que la moyenne des returns obtenus depuis l'état s (auquel on est arrivé par un chemin p donné) tendra vers la valeur $V^\pi(s, p)$. On arrive à la même conclusion en faisant la moyenne des returns obtenus par triplettes état-action-chemin : cette moyenne tend, à l'infini, vers la valeur de $Q^\pi(s, a, p)$.

Naturellement, un nombre infini d'observations est rarement accessible aussi se contente-t-on, en pratique, de prendre un grand nombre d'échantillons tout à fait au hasard et de calculer la moyenne sur cet échantillon en espérant qu'il soit proportionnellement représentatif de la population complète.

Cette méthode étant entièrement stochastique (on tire l'échantillon aussi aléatoirement que possible), elle n'est pas sans rappeler les casinos et jeux de hasard et est désormais connue sous le nom de *Méthode de Monte-Carlo*.

Il va sans dire que si notre problème comporte un grand nombre d'états et de chemins possibles, la méthode de Monte-Carlo est très peu adaptée. Une approche basée sur la paramétrisation des fonctions sera poursuivies au chapitre suivant dans le cas des problèmes de Markov.

1.4 Dilemme Exploitation - Exploration

Reprenons l'exemple de notre agent souhaitant s'échapper d'un obscur donjon. Notre agent a, au cours de son passé, tenté de sortir par une fenêtre située au 17ème étage. Le fort reward négatif qui s'en est suivi incite notre agent à se tenir désormais loin des fenêtres. Mais toutes les fenêtres ne sont pas au dix-septième étage et il peut être utile de sauter par une fenêtre si, par exemple, celle-ci est au rez-de-chaussée.

D'un côté, nous devons donc tenter de choisir des actions sûres, qui rapportent un reward positif (exploitation de la solution) et de l'autre nous souhaitons ne pas passer à côté d'une solution meilleure en terme de reward et nous devons donc prendre certains risques pour explorer l'inconnu (explo-

ration).

De manière plus rigoureuse, l'exploration correspond à l'association d'une distribution de probabilités à l'ensemble des actions possibles de chaque état.

Trouver la bonne balance entre exploitation et exploration est typique d'un problème de Reinforcement Learning et il est primordial de ne pas le perdre de vue lorsque nous implémenterons la solution.

1.5 En résumé

Dans cette section, nous avons donc introduit les principes de base du Reinforcement Learning.

Tout d'abord, il est important de distinguer explicitement l'agent de son environnement. Un agent est une entité ayant une connaissance partielle, complète voire déformée de l'environnement. En fonction des informations de l'environnement, l'agent va choisir une action qui va lui permettre d'interagir avec l'environnement. Notons que l'agent fait lui-même partie intégrante de l'environnement. Traditionnellement, on considère que les appareils de perception sont eux-mêmes partie intégrante de l'environnement mais pas de l'agent. Ainsi, dans le cas d'un robot dans un bâtiment, on considère que l'agent est le cerveau électronique du robot. Les bras, caméras et autres senseurs font partie de l'environnement extérieur¹.

En Reinforcement Learning, lorsqu'un agent effectue une action, l'environnement s'en trouve modifié et passe d'un état s à un état s' . En fonction de la modification, un reward est envoyé à l'agent. L'agent peut, de cette manière, évaluer la justesse de l'action qu'il vient d'effectuer.

On peut donc définir la policy optimale π comme la suite d'actions effectuées par un agent en choisissant, à chaque fois, l'action qui permettra de maximiser le reward total obtenu en fin de partie.

¹Cette convention à priori peu intuitive est en fait parfaitement justifiée afin de tenir compte des erreurs de mesure. Le déplacement d'un bras robotique ou l'image d'une caméra ne sont jamais en effet parfaitement modélisés : il existe une différence entre la réalité et l'information reçue/envoyée par le robot. En déplaçant ces erreurs en dehors de l'agent, l'aspect imparfait et donc probabiliste de l'information est partie intégrante de l'environnement et pourra être traité, par exemple dans le cas des jeux de Markov.

Nous avons de ce fait pu définir la valeur d'un état $V(s)$ ainsi que la valeur d'une action a effectuée à l'état s comme étant $Q(s, a)$. Le coeur du problème se résume donc à estimer ces deux fonctions.

Nous avons aussi introduit le dilemme de l'exploitation/exploration, dilemme central en reinforcement learning : il faut en effet s'assurer d'utiliser les données déjà apprises (exploitation) mais ne pas négliger la possibilité de découvrir des solutions meilleures (exploration).

Chapitre 2

Les jeux de Markov

2.1 La propriété de Markov

Dans la section précédente, nous sommes volontairement restés obscurs sur la notion d'état de l'environnement et la manière dont l'agent perçoit l'environnement.

Dans le cadre de ce mémoire, nous nous concentrerons uniquement sur la prise de décision. Cette prise de décision se basera sur un signal nous informant de l'état de l'environnement. La construction de ce signal ne sera pas traitée dans ce mémoire.

Il est évident que le signal ne pourra pas toujours contenir toutes les informations nécessaires à la résolution du problème. Un agent jouant à Stratego, par exemple, ne recevra pas d'informations sur les pièces adverses qui n'ont pas encore été découvertes. Certaines informations restent donc cachées!

D'un autre côté, il est inadmissible que notre agent se permette d'oublier des informations. Si, dans le jeu de Stratégo, notre agent a envoyé un éclaireur sur une bombe, il sait dorénavant que la pièce en question est une bombe, même si elle est pour le moment toujours cachée.

Le signal que nous désirons donc est un signal à propos de l'environnement qui contiendra toutes les informations utiles disponibles pour prendre la décision. Dans le meilleur des cas, ce signal consistera uniquement en une représentation de la situation courante. C'est par l'exemple le cas du jeu d'échec : l'historique de la partie importe peu, la position au temps t de chaque pièce est une information complète et suffisante (exemple qui peut

être relativisé si l'on souhaite apprendre la stratégie d'un adversaire donné, auquel cas l'historique de la partie a de l'importance). Dans le cas d'une partie de Stratégo, la position actuelle des pièces sera considérée mais aussi d'autres informations comme la fonction des pièces qui ont déjà été révélées et les pièces non-révlées qui ont déjà effectué un mouvement (ne pouvant, de ce fait, être une bombe). Dans le pire des cas, notre signal devra même contenir l'historique complet de tous les états rencontrés.

Un signal tel que décrit ici qui a la caractéristique de résumer et de retenir toutes les informations utiles est un signal possédant la *propriété de Markov*.

C'est typiquement le cas de notre jeu d'échec ou d'un objet soumis à la gravité dont on souhaite connaître le mouvement. Les conditions initiales de vitesse et d'altitude n'ont pas d'influence dans le calcul d'un état futur si l'on possède l'information pour un temps t . Ces problèmes sont donc indépendants du chemin ("Path independance"), les environnements possédant la propriété de Markov. Ce n'est clairement pas le cas du Stratégo. L'environnement du Stratégo ne possède pas la propriété de Markov. Néanmoins, il est très simple de concevoir, pour cet exemple, un signal possédant une telle propriété (à savoir un signal qui, en plus de la position des pièces, fournirait la valeur des pièces déjà découvertes au cours du jeu ainsi que les pièces déjà capturées).

Formalisons donc la propriété de Markov dans le cadre du Reinforcement Learning appliqué à un problème discret. Considérons, dans un cas extrêmement général, la probabilité du reward qui nous sera retourné au temps $t + 1$ après avoir effectué l'action a_t dans l'environnement s_t :

$$P_r\{s_{t+1} = s'; r_{t+1} = r | s_t; a_t; r_t; s_{t-1}, a_{t-1}; r_{t-1}; \dots; r_1; s_0; a_0\} \quad (2.1)$$

et ce pour tout s', r , et toutes les valeurs possibles des événements antérieurs : $s_t, a_t, r_t, \dots, r_1, s_0, a_0$.

Par contre, si le signal possède la propriété de Markov, alors la réponse au temps $t + 1$ ne dépend uniquement que de l'état au temps t , la probabilité du reward étant alors tout simplement définie par

$$P_r\{s_{t+1} = s'; r_{t+1} = r | s_t; a_t\} \quad (2.2)$$

et cela pour s', r, s_t, a_t .

On pourrait résumer en disant que la représentation d'un état possède la propriété de Markov si et seulement si les équations 2.1 et 2.2 sont égales

pour tout s', r et pour tout historique possible des a_t, s_t, r_t .

Par simple itération, il apparaît donc qu'à partir d'une représentation markovienne d'un état, on peut en déduire tous les états et rewards futurs (à l'approximation probabiliste près). Une représentation markovienne est donc la meilleure base pour choisir la prochaine action. Disposant d'une représentation markovienne, tout élément supplémentaire est donc superflu. On comprend donc aisément l'importance que revêt une telle représentation dans notre problème de prise de décision (notons que le passage à un problème continu se fait en transformant les sommes en intégrales et les probabilités en densité de probabilité).

2.1.1 L'approximation markovienne

On objectera qu'il faut disposer d'une telle représentation et, si nécessaire, pouvoir prouver que nous avons bien une représentation markovienne.

En pratique, il apparaît que même une représentation non-markovienne peut être approximée et utilisée comme disposant tout de même de la propriété de Markov avec un certain succès [Sutton & Barto, 1998]. Il paraît intuitif que dans beaucoup d'environnements naturels, l'influence d'un état particulier est inversement proportionnelle à son ancienneté, les états et les actions les plus récentes ayant le plus d'influence. En ce sens, l'hypothèse markovienne est donc un simple cas particulier où l'on ne considère que le dernier état.

Reprenons l'exemple du Stratégo en définissant un état comme la position des pièces et leur qualité si la pièce a été découverte au moins une fois. Cet état n'est pas Markovien car nous avons perdu, au cours du temps, l'information des pièces non-découvertes s'étant déplacées (et dont on peut donc être sûr qu'elles ne sont pas des bombes). La probabilité calculée par l'équation 2.1 est donc plus précise que celle calculée par l'équation 2.2.

Malgré cela, il semble naturel que considérer notre état comme markovien et lui appliquer les méthodes de Reinforcement Learning a de grandes chances de succès dans la prédiction des états futurs et des actions idéales à effectuer.

Le sur-coût nécessaire pour obtenir un signal markovien (lorsque cela est possible) n'est donc pas toujours pleinement justifié. On pourra éventuellement considérer un signal "approchant suffisamment" la propriété de Markov.

2.2 Les Markov Decision Processes

Un problème de Reinforcement Learning possédant la propriété de Markov est appelé un Markov Decision Process (MDP, Processus de Décision de Markov) [Sutton & Barto, 1998]. Un MDP est donc défini par un état, un ensemble d'actions et par la réponse du système à chacune de ces actions.

Pour tout état s et action a , la probabilité que l'état suivant soit s' est donc donné par :

$$P_{ss'}^a = P\{s_{t+1} = s' | s_t = s; a_t = a\} \quad (2.3)$$

On en déduit également la valeur du prochain reward :

$$R_{ss'}^a = R\{r_{t+1} | s_{t+1} = s', s_t = s; a_t = a\} \quad (2.4)$$

Avec ces deux valeurs, nous pouvons spécifier complètement un MDP.

2.2.1 Calcul des valeurs dans un Markov Decision Process

En posant l'hypothèse de Markov, nous simplifions grandement les équations valeur-état et valeur-action. En effet, il ne nous faut à présent plus du tout tenir compte du chemin mais uniquement de l'état présent.

Ainsi, l'équation 1.5 devient :

$$V^\pi(s) = E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\right\} \quad (2.5)$$

De la même manière, l'équation 1.7

$$Q^\pi(s, a) = E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s; a_t = a\right\} \quad (2.6)$$

Prenons à présent le temps de développer l'équation 2.5. Nous savons que pour $t = t + 1$, $\gamma^k = 1$. Nous obtenons donc :

$$V^\pi(s) = E_\pi\left\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s\right\} \quad (2.7)$$

Il apparaît de suite que l'on peut suivre ce principe et décomposer récursivement la fonction $V^\pi(s)$. Ainsi, si on se rappelle la définition 1.3, on voit que la formule 2.7 comporte le reward obtenu en $t+1$ et la formule de $V^\pi(s_{t+1})$. Nous pouvons donc réécrire $V^\pi(s)$ comme la somme de ces deux éléments en n'oubliant pas bien sûr de sommer $V^\pi(s_{t+1})$ pour tous les s' possibles et de les pondérer par leur probabilité d'être choisis dans la policy π .

Nous obtenons alors :

$$V^\pi(s) = r(s) + \gamma \sum_{s'} P(s'|s, \pi(s)) V^\pi(s') \quad (2.8)$$

Cette équation est appelée *équation de Bellman* de $V^\pi(s)$.

L'équation de Bellman est un énorme bond en avant dans la résolution de notre problème car nous avons enfin un moyen concret de calculer la valeur associée à un état.

Il est évident que nous pouvons aussi calculer l'équation de Bellman de Q^π [Sutton & Barto, 1998].

2.2.2 Recherche de la valeur optimale

Grâce à l'équation de Bellman, nous avons donc réussi à établir une formule qui nous permet de calculer objectivement la valeur V associée à un état s en suivant la policy π et cela en calculant récursivement la valeur de tous les états enfants possibles. Victoire ?

Il est bon, de temps en temps, de prendre un peu de distance. Souhaitons-nous pouvoir calculer la valeur de tous les états ? Avons-nous une policy précise ? Que du contraire ! Nous ne souhaitons en fait qu'une seule et simple chose : trouver le meilleur état successeur, celui qui nous guidera le plus efficacement à la solution.

Définissons donc $V^*(s)$, la *fonction valeur optimale*.

$$V^*(s) = \max_{\pi} V^\pi(s) \quad \text{et ce } \forall \pi \quad (2.9)$$

Dans ce cas de figure, l'équation de Bellman 2.8 se transforme donc en :

$$V^*(s) = r(s) + \max_a \gamma \sum_{s'} P(s'|s, a) V^*(s') \quad (2.10)$$

et ce $\forall a \in A$, l'ensemble des actions disponibles.

Et la policy me direz-vous? Et bien il suffit, simplement, une fois V^* calculé, de choisir à chaque état l'action qui mène au $V^*(s')$ le plus grand (“greedy policy”).

De la même manière, nous pouvons définir $Q^*(s, a)$, la fonction qui nous donnera le retour attendu en effectuant l'action a à l'état s et en suivant par après la policy optimale. Q^* s'écrit donc :

$$Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a) \quad \text{et ce } \forall s \in S, a \in A(s) \quad (2.11)$$

On peut bien entendu réécrire l'équation 2.11 selon $V^*(s)$ à partir de l'équation 2.9 [Sutton & Barto, 1998] :

$$Q^*(s, a) = E\{r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a\} \quad (2.12)$$

2.2.3 Méthode de résolution

Il existe plusieurs méthodes de résolution permettant de déduire la fonction Q^* ou au moins une approximation. Plusieurs de ces méthodes sont décrites dans [Tijms 2003].

Dans ce mémoire, nous nous intéresserons plus particulièrement à la méthode *Q-Learning*.

Le principe de la méthode Q-Learning est simple. Il consiste à maintenir un tableau de valeur $Q[s, a]$ avec une entrée pour chaque couple état/action. Cette entrée est précisément l'estimation pour $Q^*(s, a)$. Un agent peut donc, en fonction de son expérience, améliorer l'estimation contenue dans $Q[s, a]$ selon un “learning rate”. Le *learning rate* noté α , avec $0 < \alpha < 1$ est le degré d'importance accordé aux expériences nouvelles. Un α nul considère la table Q comme fixée et donc ne pouvant pas être mise-à-jour par l'expérience. Au contraire, $\alpha = 1$ signifie que toute ancienne connaissance doit être oubliée pour peu qu'une nouvelle expérience donne une valeur différente.

Dans la méthode de Q-Learning, nous allons utiliser trois fonctions principales :

- la fonction Q , décrite ci-dessus et qui associera une valeur à chaque couple état/action

- la fonction V , qui assigne à chaque état une valeur
- la fonction π , qui assigne, pour chaque couple état/action (s, a) la probabilité de choisir l'action a en étant à l'état s .

Les fonctions V et π peuvent être facilement réécrites comme fonctions de Q :

$$V(s) = \max_a Q(s, a) \quad (2.13)$$

Si a' est $\operatorname{argmax}_a Q(s, a)$, l'action pour laquelle $Q(s, a)$ est maximale en l'état s :

$$\pi(s, a) = \delta_{a, a'} \quad (2.14)$$

où δ est le symbole de Kronecker. L'équation 2.14 signifie donc que le probabilité $\pi(s, a)$ vaudra 1 si a est l'action qui maximise $Q(s, a)$ en l'état s , 0 sinon.

Après avoir effectué une action, l'expérience acquise par un agent peut être schématisée par le tuple (s, a, s', r) où :

- s est l'état de départ
- a est l'action effectuée
- s' est l'état obtenu après avoir effectué l'action
- r est le reward obtenu pour cette action

La règle du Q-Learning de mise à jour de la table Q est alors :

$$Q[s, a] = (1 - \alpha)Q[s, a] + \alpha(r + \gamma V(s')) \quad (2.15)$$

Remarquons que de par l'équation 2.13, l'équation 2.15 est récursive. γ est le facteur d'amortissement de l'expected reward.

Il est démontré que, dans un environnement comportant un seul agent, la méthode du Q-Learning fait converger la table Q vers la fonction optimale Q^* avec une probabilité de un [Littman 2000].

2.3 Les jeux de Markov

Un Markov Decision Process est un problème dans lequel un seul agent agit de façon stochastique. Tous les autres agents ont des comportements entièrement déterministes. Si plusieurs agents ont des comportements stochastiques, on parle alors de *Markov game*.

[Littman 2000] définit un jeu Markov par un ensemble S d'états et une collection d'ensemble d'actions A_1, A_2, \dots, A_n , chaque agent possédant son

propre ensemble d'actions. Chaque agent possède aussi sa propre fonction de reward R_i . Par opposition, [Littman 2000] précise qu'un MDP simple n'est défini que par un ensemble S d'états, un ensemble A d'actions et un ensemble R de rewards.

[Littman 1994] précise que, contrairement à un MDP, dans un Markov game il n'y a pas nécessairement une policy fixe qui serait optimale. En effet, l'optimalité dépend des mouvements des autres agents! La solution généralement retenue est, dans ce cas, d'évaluer ce qui serait pour nous la pire action que les autres agents puissent faire et de réagir en conséquence : c'est le principe de l'algorithme minimax [Russel & Norvig, 2003]. Cette solution tend à être relativement conservatrice et à privilégier les matches nuls.

Il est aussi particulièrement intéressant de remarquer qu'une stratégie optimale dans un jeu de Markov peut être stochastique! Cette affirmation peut paraître étonnante à première vue mais [Littman 1994] donne l'exemple du jeu pierre, papier, ciseau. À ce jeu, n'importe quelle stratégie déterministe peut être découverte et donc battue. Une policy idéale est donc stochastique.

[Littman 1994] montre expérimentalement que la technique de Q-Learning appliquée au MDP peut se transposer directement et avec succès à un Markov game.

2.4 En résumé

Dans cette section, nous avons défini la propriété de Markov. Un signal est dit avoir la propriété de Markov si son état actuel contient toutes les informations nécessaires pour déduire l'état actuel d'un environnement et en déduire son futur. Si on perçoit un environnement via un signal de Markov, il n'est donc pas nécessaire de retenir un historique complet des états rencontrés précédemment.

Nous avons vu qu'un problème de Reinforcement Learning possédant la propriété de Markov est appelé un Markov Decision Process (MDP). Dans ce genre de problème, il n'est donc pas nécessaire de retenir l'historique des coups précédents. Dans ce cas, il existe des méthodes pour calculer $V(s)$ et $Q(s, a)$. Nous en avons déduit l'équation de Bellman.

La méthode qui nous intéresse spécialement est appelée *Q-Learning*. Cette méthode consiste à remplir, au fur et à mesure, la matrice $Q(s, a)$ en fonction

des expériences acquises. Au fur et à mesure, la matrice $Q(s, a)$ ainsi calculée tendra à se rapprocher de $Q^*(s, a)$.

Cependant, le problème n'est un MDP aux yeux de l'agent que si tout ce qui fait partie de l'environnement se comporte de manière déterministe. Si un ou plusieurs autres agents agissent de manière probabiliste, nous avons alors affaire à un Markov Game. Dans un MDP, on s'intéresse donc à un agent seul qui interagit avec un environnement donné et tente d'arriver à un but précis. Dans un Markov Game, plusieurs agents ont des objectifs parfois antagonistes et suivent des policy stochastiques.

Chapitre 3

Le problème du chat et de la souris

3.1 Définition du problème

À titre d'exemple, le problème étudié et implémenté dans ce mémoire est un problème connu traditionnellement dans la littérature sous le terme Pursuit-Evasion [Hespanha et al, 1999].

Un ou plusieurs agents poursuivants, appelés ici “chats”, doivent attraper un ou plusieurs agents fugitifs, appelés souris. Ce type de problème se singularise des problèmes de type Search and Rescue dans le sens où l'agent fugitif cherche à tout prix à éviter les poursuivants.

Le problème chat-souris se joue au tour par tour : chaque agent effectue une action et doit attendre que tous les autres agents aient effectué une action avant de pouvoir exécuter la suivante. On considère donc un temps discret : chaque action correspond à une unité de temps $t \in \tau := 1, 2, \dots, T$. La réflexion et les observations sont considérées comme effectuées en un temps nul.

Les observations à l'instant t prennent place dans l'ensemble des observations Y et sont notées $y(t)$.

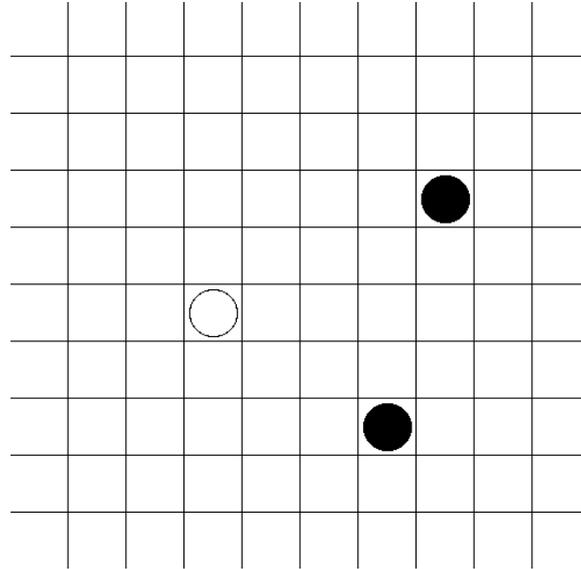


FIG. 3.1 – Exemple du problème : deux chats (cercles noirs) chassant une souris (cercle blanc) dans un espace 10x10

3.2 Espace du problème

Tout comme le temps, l'espace du problème est discret. L'espace dans lequel se déroule la partie est donc un ensemble fini et discret de cellules $\chi := 1, 2, \dots, X$ disposées en deux dimensions. Tous les mouvements sont donc discrets et concernent le déplacement d'un nombre entier de cases, posé à l'unité par défaut (voir figure 3.3).

Dans notre cas précis, nous avons particularisé le problème à un espace torique. Il n'y a donc pas de mur ni de limite : un agent s'échappant par une frontière de l'espace apparaîtra automatiquement sur la frontière opposée.

3.2.1 Calcul de la distance

Chaque espace doit définir sa propre notion de distance qui est une mesure effectuée entre deux agents. L'on pourrait ainsi imaginer une distance binaire : 0 si les deux agents occupent le même point de l'espace (ce qui est possible vu la discrétisation des états), 1 sinon.

Remarquons que cette notion n'est pas aussi simpliste que l'on pourrait le

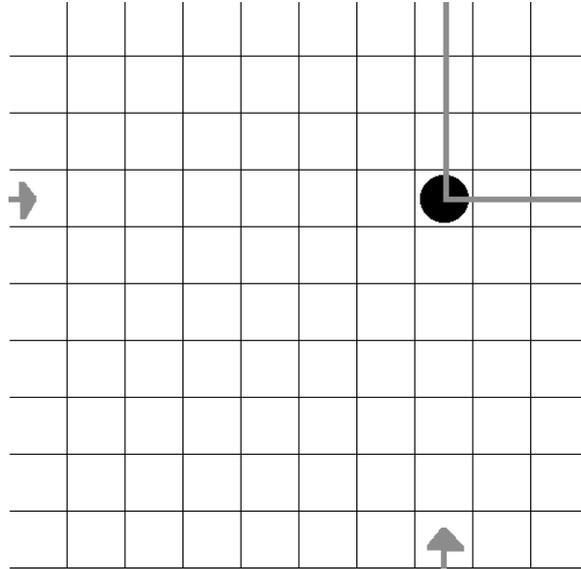


FIG. 3.2 – Illustration de deux mouvements possibles dans un espace torique

croire à première vue. Ainsi, si on définit le reward obtenu par un agent chat comme l’opposé de la distance du chat à la souris une fois l’action effectuée, on obtient bien un reward qui nous permettrait de résoudre le problème via une méthode de Reinforcement Learning.

Cependant, cette distance binaire est en pratique peu efficace : tant que le chat n’aura pas rencontré de souris, il n’aura aucune idée de la valeur de ses actions et son apprentissage sera nul.

Nous définirons donc formellement la distance entre l’agent 1 et l’agent 2 comme le nombre d’actions nécessaires à l’agent 1 pour occuper la place de l’agent 2, celui-ci ne se déplaçant pas. Remarquons que dans un espace vide et infini, cela revient à calculer la distance de Manhattan si les agents ne sont pas autorisés à utiliser les diagonales.

Il faut relever deux points importants dans notre définition :

- Notre distance tient compte de l’aspect torique de l’espace. Deux agents situés face-à-face sur des frontières opposées seront à une distance effective de 1.
- La distance tient compte d’éventuels obstacles et n’est donc pas “à vol d’oiseau”.

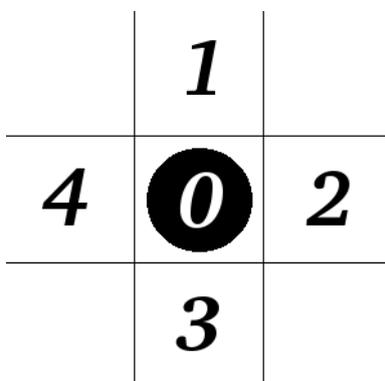


FIG. 3.3 – Mouvements possibles d'un sous-agent

3.3 L'agent chat

Un chat est un agent dont le but est d'attraper l'agent de type souris. En clair, cela signifie effectuer une série d'actions qui mènerait à un état final où la distance chat-souris serait nulle.

Au niveau de la perception, un chat possède une acuité parfaite : à tout moment, il connaît la position de tous les agents dans l'espace. L'information sur l'état de l'environnement est donc complète et correcte.

Chaque chat est symbolisé dans notre exemple par un disque noir. Un chat peut effectuer 5 actions, actions illustrées sur la fig 3.3.

- Action 0 : rester immobile
- Action 1 : Up, avancer d'une case vers le Nord
- Action 2 : Right, avancer d'une case vers l'Est
- Action 3 : Bottom, avancer d'une case vers le Sud
- Action 4 : Left, avancer d'une case vers l'Ouest

Remarquons que notre chat ne peut donc pas se déplacer en diagonale. Ce choix est purement pratique : en réduisant le nombre d'actions possibles, nous diminuons la consommation mémoire de notre implémentation.

Il s'agit donc d'un choix dans la définition de notre problème. Le problème où les agents pourraient se déplacer en diagonale est différent mais pourra être traité de manière identique.

3.3.1 Coopération entre les chats

La particularité de notre problème est que nous pouvons mettre en jeu plusieurs chats. Dans ce cas, plusieurs possibilités sont envisageables.

Dans la solution la plus simple, on se contente simplement de rajouter chaque chat comme un agent indépendant. Il n'y a ni antagonisme, ni coopération. Chaque chat se comporte juste comme si les autres chats n'existaient pas. Tout au plus se contente-t-il d'observer les réactions de la souris causées par les autres chats.

On peut aussi éventuellement imaginer un antagonisme. Dans ce cas de figure, un chat devrait attraper la souris avant les autres. Cela signifie donc prendre en compte les mouvements des autres chats dans l'établissement de la solution.

Le cas de la coopération est sans conteste le plus intéressant. Dans cet exemple, les chats font preuve d'un altruisme parfait et considèrent comme un succès égal le fait d'attraper eux-même la souris ou bien que celle-ci soit attrapée par un autre chat.

Le tout est de savoir comment prendre en compte cette coopération.

3.3.2 Agents et sous-agents

Il est tout à fait possible d'imaginer des mécanismes de coopérations entre agents [Hespanha et al, 1999].

Cependant, le fait que chaque chat dispose d'une vision globale et complète de l'environnement permet d'envisager une autre solution : un agent unique contrôlant les chats. Nous avons donc un et un seul agent de type chat. Cet agent est constitué de n sous-agents, chacun représentant un chat.

Dans le cas de notre problème, nous pouvons alors définir précisément la notion d'agent : un agent est un ensemble de sous-agents ayant un but et des observations en commun. Les sous-agents sont indissociables et interchangeables. Les sous-agents sont donc des entités unitaires capables d'effectuer un mouvement. L'ensemble des mouvements de chaque sous-agent d'un agent a au temps t est appelé l'*action* de a au temps t .

Dans notre cas, il apparaît donc clairement que l'agent de n sous-agents dispose de 5^n actions possibles.

Cette façon de procéder permet d'obtenir une coopération implicite sans complexité ajoutée.

Cependant, on remarque que cela se traduit aussi par un manque de finesse dans l'analyse d'une action. Ainsi, si un sous-agent chat effectue une très mauvaise action (tenter de rentrer dans un mur, s'éloigner de la souris) et qu'un second effectue une bonne action (s'approcher de la souris), seule l'action globale sera jugée et, selon les pondérations, sera négative ou positive. L'information selon laquelle ce que le premier chat a fait est mal et le second bien est donc perdue.

D'un autre côté, si la coopération est indispensable, cette information était-elle utile ? Pour faire démarrer une voiture, c'est très bien d'enlever le frein à main et d'accélérer, mais est-ce une bonne chose si personne n'a mis le contact ?

Cette approche, que nous avons choisie pour la suite de ce mémoire, est donc particulière et ne se prête qu'à une catégorie particulière de problèmes.

3.3.3 Apprentissage et reward

L'agent chat va donc devoir attraper la souris mais n'a aucune connaissance a priori. C'est d'ailleurs là que réside la force du Reinforcement Learning.

Cas d'un Markov Decision Process

Si le mouvement de la souris est déterministe, le problème se résume à un Markov Decision Process. L'on peut appliquer différentes méthodes dont par exemple la méthode dite d'*itération de valeur* [Littman 1994].

La valeur d'un état, notée $V(s)$, est l'expected reward total que l'on peut atteindre en suivant une policy idéale depuis l'état s . Rappelons que cet expected reward est amorti. Un état avec une grande valeur $V(s)$ est donc un état à favoriser par rapport à un état de moindre valeur. La qualité de la paire état-action, notée $Q(s, a)$, est l'expected reward total obtenu par une

policy qui, à l'état s , choisirait l'action a puis, de là, continuerait sur une policy idéal.

De ces définitions, on peut déduire trivialement :

$$V(s) = \max_{a \in A} Q(s, a) \quad (3.1)$$

Posons $R(s, a)$ le reward direct obtenu en effectuant l'action a depuis l'état s . On peut donc écrire, sachant qu'effectuer l'action a à l'état s nous amène à l'état s' :

$$Q(s, a) = R(s, a) + \gamma V(s') \quad (3.2)$$

où γ est bien sûr le facteur d'amortissement. Avec l'équation 3.2, Q décrit bien l'expected reward total obtenu en effectuant a en s puis en suivant une policy idéale depuis s' , cela en utilisant récursivement l'équation 3.1.

Cependant, un problème de taille se pose : comment savoir avec précision dans quel état s' va nous amener l'action a ? Rappelons que l'adversaire, la souris dans notre cas, peut aussi se déplacer. Cette stratégie ne serait donc efficace que pour une souris immobile.

Posons donc $T(s, a, s')$, la probabilité d'obtenir l'état s' en effectuant l'action a à l'état s . Le calcul de $Q(s, a)$ nécessitera donc de tenir compte de chaque s' possible en le pondérant par sa probabilité :

$$Q(s, a) = R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') V(s') \quad (3.3)$$

Connaissant $Q(s, a)$, notre agent peut se satisfaire d'une stratégie *greedy*, c'est-à-dire se contenter de choisir, à l'état s , l'action a pour laquelle $Q(s, a)$ est maximale. Q étant par définition la somme des rewards futurs, cette stratégie est optimale.

Cas d'un jeu de Markov

Si notre souris a un comportement stochastique et non déterministe, le problème devient un jeu de Markov. [Littman 1994] propose deux solutions pour ce genre de problème. Le premier est un croisement de minimax et de Q-Learning, appelé à juste titre minimax-Q. Il tient compte du pire mouvement que pourrait effectuer la souris.

Le second est une simple transposition du Q-Learning sans tenir compte de la stochasticité de la souris, méthode qui donne des résultats étonnamment bons.

Dans un premier temps, c'est cette seconde approche que nous avons choisi de suivre dans notre problème. En effet, lorsque la souris se trouve à une distance assez grande des chats, son mouvement influence peu. Les chats se contentent de se rendre dans la direction de la souris et ce, quelle que soit son action. À un niveau rapproché, le choix des actions d'une souris est plus que restreint si elle ne veut pas se jeter dans la gueule du loup¹. Si la souris ne dispose que d'une seule action possible permettant de s'échapper, la stochasticité ne rentrerait pas en jeu. Si la souris dispose de plusieurs actions, il est probable que toutes ces actions aient un poids égal. L'algorithme minimax ne pourrait donc rien nous apprendre de plus que le simple Q-Learning. Enfin, les résultats expérimentaux de [Littman 1994] tendent à démontrer que le simple Q-Learning sans stochasticité est plus performant que minimax-Q.

3.4 L'agent souris

L'agent souris, symbolisé par un disque blanc, jouit exactement des mêmes propriétés que le chat : vision globale de l'espace et cinq actions possibles. Cependant, nous ne considérerons pas ici le cas de plusieurs souris. En effet, avoir plusieurs souris demanderait alors de spécifier le problème de manière différente en terme de coopération, d'objectif, etc.

Notre souris a donc pour but d'échapper aux chats. Plusieurs solutions sont envisageables comme un minimax classique. Nous avons cependant préféré développer un algorithme personnalisé intitulé *best-escape*

3.4.1 L'algorithme best-escape

Le principe de l'algorithme best-escape est très simple. Il consiste tout d'abord par refuser tout mouvement qui jetterait la souris dans les griffes d'un chat ou tout mouvement qui conduirait la souris dans le voisinage immédiat d'un chat (nous entendons par là la case adjacente). Ces deux actions seraient en effet synonymes d'une mort certaine et d'une victoire sans saveur pour les chats.

¹Du chat bien sûr !

Pour chacune des actions restantes, on calcule l'*escape value*, la somme des carrés des distances du point d'arrivée de l'action avec chacun des chats présents dans le jeu. Le but étant évidemment de maximiser cette valeur.

Dans la version déterministe de l'algorithme, on choisit alors l'action donnant une *escape value* la plus grande possible. Si plusieurs actions ont la même *escape value*, on prend la première de la liste. Le choix est donc entièrement déterministe et peut être prédit pour chaque situation donnée.

Si la souris obéit à cette règle, nous sommes donc bien dans le cas d'un MDP. Cependant, on peut rendre l'algorithme best-escape stochastique en assignant à chaque action une probabilité proportionnelle à son *escape-value*. Posé ainsi, nous nous retrouvons dans le cas d'un jeu de Markov.

Algorithme 1: Algorithme best-escape stochastique

- (1) $s = \text{etat}, d_{total} = 0$
- (2) available actions = actions for souris(s)
- (3) for a in available actions :
- (4) $d(a) = \min_{chat} \text{distance}(\text{souris}(s, a), chat)$
- (5) if $d(a) \leq 1$ then remove a from available actions
- (6) $d_{total} = d_{total} + d(a)$
- (7) choose action a with probability $d(a)/d_{total}$

3.5 Obstacles

Un troisième type d'agent que nous pouvons utiliser sont les pierres. Parler d'agent pour les pierres est un peu exagéré, il est vrai, ces agents n'ayant aucune action possible (si ce n'est de rester en place). Ils sont donc particulièrement déterministes.

Les obstacles que constituent les pierres peuvent changer radicalement la morphologie du jeu. Il est à remarquer un fait amusant : si la distance dans notre espace est implémentée comme étant une distance de Manhattan ou tout autre distance euclidienne, on constate que les pierres sont invisibles. Les agents vont en effet n'en tenir compte qu'une fois le nez dessus. Un chat pourra se ruer à la poursuite d'une souris avant de s'écraser le nez. Ce concept peut être particulièrement intéressant pour simuler un espace inconnu à explorer. Les agents recevraient un reward négatif en se cognant contre un mur.

Ce problème est évité si la distance est implémentée selon la définition donnée plus haut dans ce chapitre.

3.6 Récapitulation : agents et environnement

Nous avons donc défini un environnement dans lequel coexistent trois types d'agents :

- L'agent chat, dont le but est d'attraper la souris.
- L'agent souris, dont le but est d'échapper au chat.
- L'agent pierre, dont le but est de ne pas bouger, quoiqu'il puisse arriver.

Chaque agent possède un nombre de sous-agents compris entre 0 et n . Au sein d'un agent, les sous-agents sont considérés comme parfaitement équivalents et interchangeable. Il est donc important de remarquer qu'une situation au sein de l'environnement reste parfaitement identique si l'on intervertit deux sous-agents d'un même agent.

Introduisons aussi deux définitions :

- *Mouvement* : le mouvement d'un sous-agent est le choix d'un déplacement donné. Ce mouvement peut-être nul auquel cas le sous-agent reste dans la même position.
- *Action* : une action d'un agent est l'ensemble des mouvements de tous les sous-agents de cet agent durant une étape de temps t .

Ainsi, la phrase "Le chat a mis 100 actions pour attraper la souris" signifie que chaque sous-agent de l'agent chat a effectué 100 mouvements (éventuellement nuls).

Dans ce mémoire, nous nous concentrerons sur les cas où le nombre de sous-agents de l'agent souris est l'unité, celui de l'agent pierre est nul et celui de l'agent chat est deux ou trois.

3.7 Analyse du problème

Avant de nous lancer tête baissée dans le problème, il est peut-être utile de tenter de l'analyser avec un peu de recul.

3.7.1 2 chats et une souris

Il nous semble évident qu'un chat seul n'a pas la moindre chance d'attraper une souris si celle-ci est tant soit peu intelligente. Mais qu'en est-il de deux chats ?

En exploitant la symétrie du problème, on peut constater que deux chats ne peuvent entourer une souris que de 6 façons différentes, ces 6 façons étant

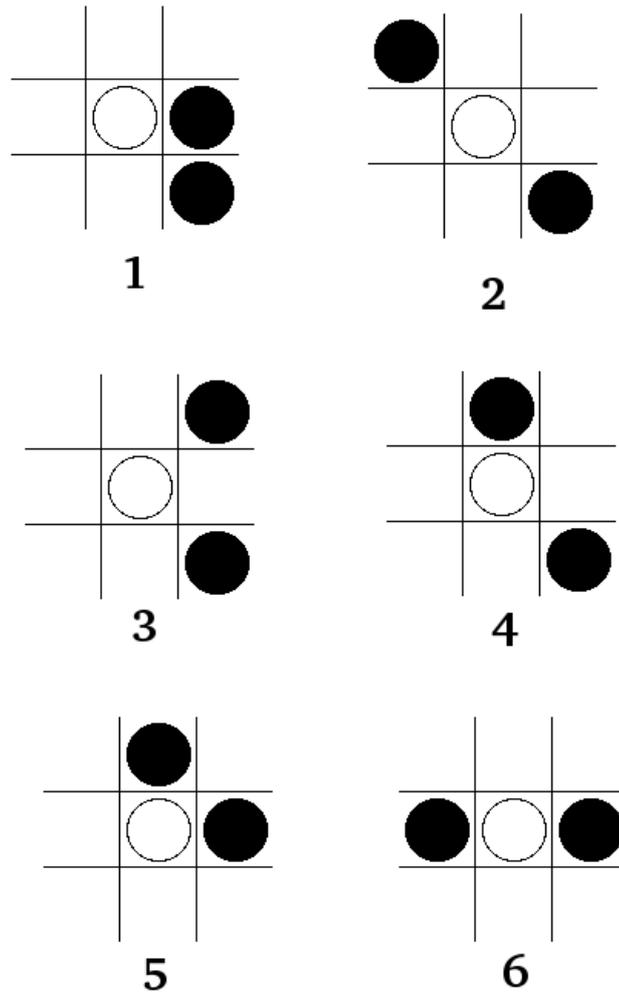


FIG. 3.4 – Les 6 façons pour 2 chats d'encercler une souris

illustrées sur la fig.3.4. Or, aucune de ces façons ne permet d’attraper à coup sûr la souris. Seule la façon 2 bloque toutes les issues de la souris : il lui suffit alors d’effectuer l’action 0 (rester immobile), ce que garantit l’algorithme best-escape.

De manière plus mathématique, on se rend compte que la géométrie du problème ne permet à chaque chat que de bloquer, au mieux, deux actions de la souris. Au total, les chats peuvent donc bloquer 4 actions de la souris qui en dispose de 5.

Il est aussi intéressant de remarquer que même dans un espace 3x3, la solution n’est pas aussi triviale qu’on pourrait le croire. En effet, dans un espace 3x3, seule la configuration 5 permet de capturer la souris.

Contrairement à ce que l’on aurait pu croire intuitivement, l’avantage est donc très nettement à la souris.

Fatigue de la souris

Afin de rendre la simulation néanmoins intéressante, nous avons introduit un facteur de fatigue chez la souris. Ce paramètre, très simple, introduit en fait la probabilité que la souris n’effectue aucune action à un moment donné et cela sans égard au reste de l’environnement. On peut voir ça aussi comme le fait que la souris soit figée de peur. Ce paramètre a pour but de donner un petit coup de pouce aux chats. La souris aura une probabilité non-nulle de ne pas choisir d’action et se reposer.

3.7.2 3 chats et une souris

Comme nous l’avons vu, le problème de deux chats et une souris est impossible sans “tricher” (la fatigue de la souris). Nous allons donc aussi tester le cas où trois chats tentent d’attraper une souris.

Dans le cas de 3 chats et une souris, le problème devient parfaitement possible. Pour s’en convaincre, il suffit de dessiner une troisième chat sur les exemples de la figure 3.4. Mais contrairement à ce qu’on pourrait croire, le problème est loin d’être trivial. Les chats doivent en effet être parfaitement coordonnés pour ne laisser aucune chance à la souris. De plus, le nombre d’actions possibles pour l’agent chat devient alors :

$$C_{n-1}^3 \tag{3.4}$$

Où n est le nombre de cases de l'espace². Le fait d'avoir trois sous-agents augmentent donc considérablement le "branching factor" de l'arbre des actions de l'agent.

3.8 Dilemme exploration-exploitation

Comme nous l'avons vu au chapitre 1, l'un des gros défis du Reinforcement Learning est de concilier exploration et recherche d'une éventuelle meilleure solution avec exploitation et utilisation effective des solutions déjà trouvées.

Dans le cas de la poursuite chat-souris, on peut résumer l'exploration-exploitation au problème suivant : lorsque l'agent chat se trouve dans une situation déjà rencontrée, il a deux types d'actions possibles :

1. Refaire une action déjà tentée en pareille circonstance (et donc avec une grande probabilité d'obtenir un reward fort semblable ou du même ordre).
2. Tenter une action non-tentée jusqu'à présent en pareil cas.

Il apparaît clairement qu'une exploration nulle ne pourra mener nulle part. En effet, à chaque état précédemment rencontré, le chat réitérera inlassablement la même action qui peut très bien être sub-optimale, celle-ci ayant été choisie aléatoirement la première fois. Le chat peut même en être réduit à boucler. Une exploration intelligente est donc primordiale !

3.8.1 Exploration stochastique fixe

Une approche on ne peut plus simple consiste à fixer un ratio d'exploration fixe. À chaque étape, l'agent chat aura une probabilité fixée d'explorer. Dans le cas contraire, on passera alors à la phase d'exploitation et on choisira, parmi les actions connues, une de celles ayant le reward le plus élevé (greedy policy).

Cette approche est celle suivie dans [Littman 1994]. Littman suggère une valeur de 0.2 pour la probabilité d'exploration. Remarquons que l'exploration n'implique pas nécessairement d'effectuer une action non-tentée ! Il s'agit bien au contraire de tenter une action au hasard. Si l'action avait déjà été tentée, l'apprentissage serait donc renforcé et la table des états-actions serait

²On choisit donc $n - 1$ car la souris occupe forcément une des cases

mise à jour suivant l'équation 2.15 comme détaillé au Chapitre 2.

Cette approche semble cependant inappropriée dans le cas d'une poursuite chat-souris. En effet, il semble au départ préférable de favoriser une période d'apprentissage en raison du grand nombre d'états possibles. La solution de Littman sera plus adaptée dans un problème avec un petit nombre d'états.

D'autre part, augmenter le coefficient d'exploration pourrait se révéler désastreux. À l'extrême limite, on obtiendrait un comportement entièrement aléatoire.

3.8.2 Exploration stochastique décroissante

La méthode de Littman ne nous a donc pas semblé appropriée au problème de la poursuite chat-souris. Les expériences devaient d'ailleurs corroborer cette hypothèse.

Il nous est alors venu à l'esprit d'effectuer une période d'apprentissage avec un degré d'exploration élevé voire proche de l'unité puis de passer à une phase d'exploration pure avec un degré d'exploration restreint.

Cette approche, cependant, se révèle sub-optimale sous plusieurs aspects. Il n'est en effet pas toujours possible de séparer clairement apprentissage et exploitation. Ensuite, on veut continuer à apprendre s'il s'avère que la période d'apprentissage n'a pas été suffisante eu égard à la taille du problème. Enfin, et c'est probablement le point le plus important, une exploration purement aléatoire serait sub-optimale par essence. En effet, il est plus que probable que suivre à la racine une branche de l'arbre des solutions qui s'est révélée particulièrement mauvaise n'apportera pas de solution miracle. Par contre, explorer et raffiner la connaissance d'une sous-branche déjà connue comme favorable peut se révéler utile. En clair : il y a de fortes chances que la solution idéale soit plus proche de la meilleure solution actuellement connue que d'une solution quelconque. Un coefficient d'exploration trop élevé aurait tendance à explorer "dans tous les sens" là où un coefficient d'exploration plus faible concentrerait l'exploration dans les zones déjà connues comme bonnes (tout en ne rejetant pas totalement le reste).

Une réponse possible consiste à utiliser une exploration de type *stochastique décroissante*.

L'exploration stochastique décroissante est, comme son nom l'indique,

une amélioration de la solution proposée par Littman. Nous allons bel et bien partir d'un coefficient d'exploration donné mais, à chaque itération (et donc à chaque fois que l'agent effectue une action), nous allons multiplier ce coefficient par un facteur donné D_{explor} :

$$Explor(t + 1) = Explor(t) * D_{explor} \quad (3.5)$$

Cette méthode nous permet donc d'exercer un contrôle beaucoup plus fin sur le degré d'exploration. Alors qu'au départ, nous privilégierons une exploration importante, afin que le chat explore son environnement, nous allons au fur et à mesure du jeu favoriser l'exploitation. L'exploitation est donc inversement proportionnelle au degré de connaissance de l'environnement.

Cette approche nécessite néanmoins de fixer des valeurs pour $Explor(t)$ et pour D_{explor} . Dans notre cas, nous tenterons une approche expérimentale avec plusieurs valeurs pour ces paramètres.

3.8.3 Entropie et degré d'exploration

Une méthode plus rigoureuse est proposée dans [Achbany et al, 2006]. [Achbany et al, 2006] introduit le concept de *degré d'exploration* d'un état s , noté E_s , définit comme l'entropie de Shannon liée à la distribution de probabilité de l'ensemble des actions possibles pour cet état.

$$E_s = - \sum_{a \in U(s)} \pi_s(a) \log \pi_s(a) \quad (3.6)$$

E_s peut être interprété comme la mesure de l'incertitude du choix à effectuer. $E_s = 0$ signifie donc une incertitude nulle et donc qu'il existe $\pi_s(a) = 1$, avec a le choix optimal. Au contraire, si tous les choix ont une probabilité égale, on a une incertitude maximale et $E_s = E_{\max}$.

Le problème se réduit donc, dans un état donné, à maximiser le reward attendu tout en maintenant un degré d'exploration en dessous d'une valeur donnée.

Dans le cas d'une exploitation stochastique, la probabilité de choisir une action a à l'état s est donné par :

$$\pi_s(a) = \frac{Q(s, a)}{\sum_{i \in A(s)} Q(s, i)} \quad (3.7)$$

En réécrivant l'équation 3.7 selon 3.2 et en posant $\gamma = 1$ pour la simplicité du raisonnement³, on obtient :

$$\pi_s(a) = \frac{R(s, a) + V(s'_a)}{\sum_{i \in A(s)} R(s, i) + V(s'_i)} \quad (3.8)$$

Afin de contrôler l'exploration, [Achbany et al, 2006] introduisent dans l'équation 3.8 un paramètre θ_s qui assure le contrôle de la stochasticité :

$$\pi_s(a) = \frac{\exp[-\theta_s (R(s, a) + V(s'_a))]}{\sum_{i \in A(s)} \exp[-\theta_s (R(s, i) + V(s'_i))]} \quad (3.9)$$

Notons que si $\theta = 0$, on obtient un tirage tout à fait aléatoire en pure exploration. Si $\theta = \infty$, on se retrouve alors dans le cas d'un algorithme min-max entièrement déterministe.

Il ne reste donc plus que, selon l'équation 3.6, à calculer la valeur de θ_s telle que :

$$\sum_{a \in U(s)} \pi_s(a) \log \pi_s(a) = -E_s \quad (3.10)$$

La valeur de θ_s doit être calculée pour chaque état s et prend une valeur comprise dans l'intervalle $[0, \infty]$. L'équation 3.10 garantit un degré d'exploration (entropie) fixe à chaque état. Notons que cette équation n'a malheureusement pas de solution analytique et doit être estimée numériquement.

[Achbany et al, 2006] remarque aussi que fixer $E_s = 0$ pour tous les états s se ramène aux équations de Bellman (équation 2.8) décrivant la recherche de la solution sans aucune exploration (exploitation pure). De manière semblable, si $E_s = E_{\max}$ pour tous les états s , les équations se réduisent à la situation d'exploration aveugle :

$$\pi_s = \frac{1}{n_s} \quad (3.11)$$

³[Achbany et al, 2006] donnent tout d'abord l'exemple du cas particulier d'un problème dans lequel on cherche le plus court chemin. Dans ce cas précis, γ n'a pas de sens et peut être posé à 1. De plus, le problème de maximisation de reward devient donc un problème de minimisation, ce qui est, au signe près, parfaitement équivalent. Les équations sont généralisées avec un γ arbitraire dans la suite de l'article.

où n_s est le nombre d'actions possibles à l'état s .

Cette approche, bien que rigoureuse et précise, a cependant le défaut d'être plus complexe et gourmande en temps de calcul.

3.9 En résumé

Dans ce chapitre, nous avons précisément défini le problème qui nous occupe, à savoir celui de chats cherchant à capturer une souris dans un espace discret torique.

Dans l'environnement, il est important de définir la notion de distance, du moins si l'on souhaite utiliser la distance dans le calcul du reward pour les chats.

Nous avons aussi évoqué le problème de la coopération entre les différents agents chats. La solution proposée consiste ici à ne considérer qu'un seul agent qui disposerait de plusieurs sous-agents, chacun d'eux étant un chat.

Le chat est un élément particulièrement important de notre problème. C'est en effet lui qui va devoir effectuer du Reinforcement Learning. Nous avons donc précisé les méthodes à utiliser et définit les différentes manières de concevoir un reward. La solution la plus appropriée nous a semblé l'inverse de la distance entre le chat et la souris.

Nous avons abordé le cas de la souris. En effet, pour que le problème reste intéressant, il faut que celle-ci soit suffisamment intelligente. Nous avons donc mis au point et explicité l'algorithme *best-escape*, algorithme déterministe idéal, ainsi qu'une dérivation stochastique de cet algorithme afin de nous mettre dans la situation d'un jeu de Markov.

Il a été démontré que le problème ne comporte pas de solution possible si les chats ne sont pas au minimum 3 ou l'espace réduit à une taille inférieure ou égale à 3x3. Nous avons proposé d'introduire le concept de fatigue de la souris pour garder le problème intéressant.

Enfin, nous avons évoqué 3 différentes méthodes pour traiter le dilemme exploration/exploitation propre à un problème de Reinforcement Learning. Nous avons choisi de nous concentrer sur la méthode stochastique décroissante mais il serait très intéressant de la comparer avec celle de l'entropie.

Chapitre 4

Implémentation d'un framework de Reinforcement Learning

Pour les besoins de ce mémoire, un framework de Reinforcement Learning a été entièrement développé dans le langage Python. Si ce framework a été développé spécifiquement pour le problème chat-souris, nous l'avons voulu assez généraliste afin de permettre de l'adapter ou de l'étendre aussi facilement que possible.

Trois composantes principales définissent ce framework : l'environnement, l'agent et les définitions. L'environnement et l'agent sont définis d'une manière aussi généraliste que possible. Les définitions comportent, pour chaque agent, les définitions des spécificités de chaque agent. C'est donc dans la définition du chat qu'on va trouver le coeur de notre Q-Learning.

4.1 L'environnement

L'environnement définit l'espace dans lequel vont évoluer les agents. Dans notre implémentation, cet espace est une simple matrice d'entiers. 0 signifie une case vide. Chaque autre entier signifie un agent qui s'est désigné lui-même sous cet entier. Rappelons que l'on peut avoir plusieurs sous-agents pour un seul agent et donc plusieurs fois le même entier dans la matrice. Il est tout à fait possible d'imaginer étendre cet environnement pour, par exemple, effectuer des recherches dans un espace à 3 dimensions voire plus.

Deux notions primordiales de la composante environnement sont la dis-

tance et l'identification d'un état.

4.1.1 La distance

Comme nous l'avons vu, la notion de distance joue un rôle primordial. La distance de Manhattan peut être implémentée sans difficulté même si le fait que l'espace soit torique rend l'exercice laborieux.

Cependant, rappelons que nous souhaitons une distance qui exprime *le nombre minimum de mouvements à effectuer pour gagner la place actuellement occupée par un autre agent*.

Une technique d'implémentation de ce type de distance est la simple récursivité.

Algorithme 2: Calcul de distance par récursivité

- (1) distance(a,b) :
- (2) if a == b then return 0
- (3) else :return $\min_m v(1+\text{distance}(a+mv,b))$

Cependant, on conçoit aisément que cette méthode peut se révéler fort coûteuse en temps et en espace mémoire. Il n'est donc pas inintéressant de considérer la distance de Manhattan lorsqu'on a peu ou prou d'obstacle.

Notons que, tel quel, l'algorithme 2 est une *fork bomb*. L'implémentation nécessite donc de parcourir l'arbre des distances selon une méthode *Breadth-first search* stricte, ce qui se traduit par une complexité spatiale et temporelle en $O(b^{d+1})$ avec b le branching factor et d la profondeur [Russel & Norvig, 2003].

4.1.2 Identification d'un état

Afin de pouvoir identifier un état, il est important de pouvoir lui donner un nom unique. L'idée la plus sensée consiste à prendre tout simplement le contenu de la matrice. On obtient alors un nom de type :

00000000000000000000100002000000001000000000..

Cependant, il est important de constater que la nature torique de notre espace perturbe quelque peu les conventions. Ainsi, sur la figure 4.1, les deux situations sont rigoureusement identiques !

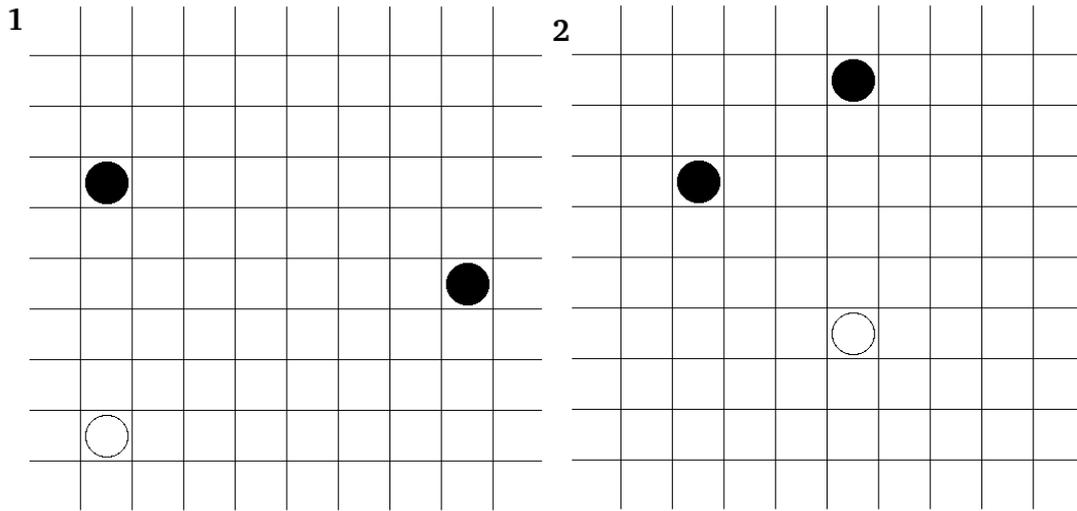


FIG. 4.1 – Deux états rigoureusement identiques par translation

Il faut donc trouver un moyen afin que les états semblables (et eux seuls) aient le même nom. Une technique simple mise en oeuvre dans ce framework consiste à utiliser comme nom le contenu de la matrice à partir de la souris et non pas à partir du coin supérieur gauche qui est purement arbitraire.

Ce faisant, nous divisons par 100 l'espace des états. C'est non seulement une très bonne chose d'un point de vue implémentation pour la mémoire vive, mais c'est surtout indispensable pour la qualité des résultats : rien qu'en faisant ce simple changement, nous avons divisé par 100 le nombre d'actions nécessaire pour l'apprentissage de la matrice Q .

Nous avons donc exploité la symétrie de translation présente dans le problème. Remarquons que le problème dispose aussi d'une symétrie de rotation et d'une symétrie orthogonale. Cependant, exploiter ces symétries est plus ardu car il faut en même temps interchanger les valeurs d'actions. Ainsi, par symétrie orthogonale, il faudrait intervertir les $Q(s, a)$ pour $a = up$ et $a = bottom$.

Remarquons aussi que la présence d'un éventuel obstacle gâche toute la symétrie et rend du coup le problème beaucoup plus complexe d'un point de vue spatial (la mémoire nécessaire est multipliée par 100) mais aussi d'un point de vue de l'apprentissage.

4.2 L'agent

Comme expliqué précédemment, la particularité de notre agent est qu'il peut consister en plusieurs sous-agents. Nous avons donc défini deux types d'objets : les agents et les subagents. Un agent contient donc un ou plusieurs subagents.

4.2.1 L'objet agent

L'agent est bien sûr primordial dans notre implémentation. Notons cependant que l'objet agent est en-lui même très généraliste : tous les éléments de l'environnement sont en effet des agents ! Chats, souris mais aussi les obstacles !

L'agent implémente plusieurs méthodes utilitaires permettant d'obtenir les positions et le nombre de subagents présents. Il existe cependant trois méthodes indispensables et qui sont propres à chaque agent :

1. *do_action* : effectue une action et modifie donc l'environnement
2. *best_move* : effectue la meilleure action selon sa policy
3. *get_nbr_actions* : renvoie le nombre d'actions possibles à un état donné

Notons que *best_move* prend en compte la policy complète : il se peut que *best_move* effectue un mouvement entièrement aléatoire si la policy a décrété qu'il fallait explorer et non exploiter. Le *best_move* n'est donc pas à comprendre comme de l'exploitation seulement.

La méthode *get_nbr_actions* n'est pas aussi triviale qu'elle le laisse supposer à première vue. En effet, si l'agent comporte plusieurs subagents, le nombre d'actions est la combinaison des mouvements possibles pour chaque subagent. Un choix d'implémentation se pose également dans la définition d'action possible : une action possible est-elle une action que l'agent peut tenter ou une action que l'agent peut réaliser ? Ainsi, dans le premier cas, face à un obstacle, l'agent peut tenter de se diriger vers l'obstacle. Il en résultera une action nulle sans modification de l'environnement. Dans le second cas, se diriger vers l'obstacle ne fait pas partie des actions possibles.

Pour le chat, le premier cas nous a semblé plus intéressant. En utilisant le Reinforcement Learning, nous avons en effet associé un fort reward négatif avec de telles actions, à charge pour le chat d'apprendre à éviter les murs. La souris, elle, est implémentée suivant la seconde possibilité.

Pour un agent chat comportant n subagents chats, la méthode *get_nbr_actions* renverra donc toujours la valeur :

$$\text{get_nbr_actions} = 5^n \quad (4.1)$$

D'un point de vue implémentation, le constructeur de l'objet prend en paramètre un objet de type *objet*¹.

Objet doit implémenter les méthodes suivantes :

1. *nbr_actions(self)* : renvoie le nombre d'actions possibles
2. *actions(self,i,agents,env)* : effectue l'action i . Agents est la liste des positions des subagents et env est l'objet environnement.
3. *find_best_move(self,env,agents)* : renvoie la meilleure action à effectuer dans l'environnement env avec les positions des subagents contenues dans agents.

Notons aussi qu'un objet agent doit comporter une variable *self.name*, un nombre entier qui doit être unique dans l'environnement.

Comme précisé plus haut, *find_best_move* prend en compte l'exploration. Il ne s'agit donc pas d'un mouvement d'exploitation pur comme le nom pourrait le suggérer.

Pour construire un agent personnalisé, il suffit donc d'implémenter ces méthodes dans un objet et de passer l'objet ainsi obtenu en paramètre au constructeur *agent*. À titre d'exemple, l'implémentation de l'objet *objet obstacle* est décrite à la figure 4.2. Dans ce cas particulier, elle est particulièrement triviale².

Il est cependant évident que l'implémentation de l'agent chat sera plus complexe, tout particulièrement la méthode *find_best_move* qui comportera le coeur de l'algorithme de Reinforcement Learning.

¹C'est malheureusement le nom utilisé dans l'implémentation

²Notons que la méthode *actions* ne doit normalement pas être appelée pour un obstacle. Il est donc encouragé de renvoyer une erreur dans l'implémentation de ce cas particulier. La méthode doit cependant exister de par la définition de l'objet.

```

class obj_stone :

    def __init__(self, pos) :
        self.name = 3

    def nbr_actions(self) :
        return 0

    def actions(self,i,agents,env):
        print "I'm a stone"

    def find_best_move(self,env,agents) :
        return 0

```

FIG. 4.2 – Implémentation de l'objet *objet* pour un obstacle

4.2.2 L'objet subagent

C'est dans l'objet subagent qu'est implémenté le mouvement lui-même. L'implémentation de subagent doit donc être construite en relation étroite avec l'environnement.

Rappelons qu'une action d'un agent est une combinaison des mouvements des subagents de cet agent. Dans tout le reste du framework, l'action est donc abstraite simplement comme un nombre entier. Il est de la responsabilité de l'agent d'assigner à chaque subagent un mouvement en fonction de l'action donnée. Mais le mouvement reste toujours un nombre entier abstrait. Le subagent a lui la responsabilité d'effectuer le mouvement et donc de modifier l'environnement.

Deux méthodes sont importantes à signaler dans subagent : *move* et *test*.

Move effectuera le mouvement et renverra le nom du subagent se trouvant sur la case où l'on s'est déplacé. La méthode *test* renvoie le nom du subagent actuellement sur la destination mais n'effectue pas l'action. En effet, rien n'empêche plusieurs agents d'occuper la même position et c'est d'ailleurs le but du chat : occuper la même position que la souris. Notons que dans le cas où la position cible est vide ou occupée par le subagent lui-même, les deux méthodes retournent 0.

```

if self.x >= self.size_x :
    self.x = self.x - self.size_x
if self.y >= self.size_y :
    self.y = self.y - self.size_y
if self.x < 0 :
    self.x = self.x + self.size_x
if self.y < 0 :
    self.y = self.y + self.size_y

```

FIG. 4.3 – Gestion d’un espace torique dans l’implémentation du mouvement

Ces deux méthodes ont donc besoin de connaître les lois de la physique propres à notre espace. C’est donc à subagent qu’est déléguée la gestion de la toricité de notre espace, gestion dont l’implémentation est visible à la figure 4.3. C’est cette partie du code qui devra être modifiée si l’on souhaite utiliser un espace obéissant à des règles différentes.

4.3 Le chat

Le chat est bien entendu une pièce centrale dans notre problème. C’est en effet au chat qu’incombe la tâche d’apprendre à attraper la souris et c’est donc dans ce dernier que nous allons implémenter le coeur du Reinforcement Learning.

Comme nous l’avons vu ci-dessus, nous avons implémenté un objet de type *objet* qui comporte les 3 méthodes suivantes :

1. *nbr_actions(self)* : renvoie le nombre d’actions possibles
2. *actions(self,i,agents,env)* : effectue l’action i. Agents est la liste des positions des subagents et env est l’objet environnement.
3. *find_best_move(self,env,agents)* : renvoie la meilleure action à effectuer dans l’environnement env avec les positions des subagents contenues dans agents.

4.3.1 Actions possibles

Dans tout le framework, une action est désignée par un nombre entier et correspond à un mouvement de chacun des subagents de l’agent concerné. Si le nombre d’actions possibles ne pose pas problème (voir équation 4.1), la

première difficulté consiste à assigner pour chaque action la suite des mouvements de chaque sous-agent.

La première implémentation consistait en un simple dictionnaire assignant, pour chaque action, les mouvements correspondants. Cependant, écrire un tel dictionnaire est fastidieux et, surtout, demande d'être modifié à chaque fois que l'on change le nombre de sous-agents !

Nous avons donc usé d'une petite astuce en décomposant le numéro de l'action par des divisions entières successives et en assignant comme mouvement le modulo du résultat ainsi obtenu. Cela est fait grâce à l'algorithme 3.

Algorithme 3: Décomposition de l'action pour obtenir les mouvements correspondants

- (1) `base = nbr_mv`
- (2) `reste = action`
- (3) `for each sub_agent :`
- (4) `mv = reste modulo(base)`
- (5) `sub_agent.move = mv`
- (6) `reste = reste div base`

Dans notre problème, `nbr_mv` est bien sûr 5 (voir figure 3.3). `Action` est le numéro de l'action et est donc un nombre compris entre 0 et $base^n - 1$. Notons aussi que `div` est une division entière. Cela revient à faire une division classique après avoir soustrait le modulo `mv` du reste.

Cette méthode n'est bien entendu utilisable que si chaque subagent possède le même nombre de mouvements et que ce nombre est constant. Une fois ces hypothèses acquises, le nombre de subagents peut être modifié à la volée sans aucun problème.

4.3.2 Reward

Comme nous l'avons vu au chapitre 1, le principe même du Reinforcement Learning repose sur la notion de reward. Le reward est propre à chaque agent et il est donc logique que cela soit le chat qui calcule le reward personnel.

Dans la fonction `reward`, nous définissons tout d'abord certains cas particuliers. Ainsi, si une action conduit le chat sur la même position que la souris, un reward infini (ou du moins très grand) est attribué. Au contraire,

si une action est bloquée par un obstacle ou un agent autre que la souris, elle se verra attribuée un fort reward négatif. Rappelons que, au départ, les chats n'ont pas de connaissance à priori. Ils ne savent donc pas que passer à travers les murs n'est pas possible.

Si nous ne nous trouvons dans aucun des cas décrits plus haut, alors il va falloir utiliser la notion de distance. L'équation 4.2 décrit la formule exacte du reward avec $d(c, m)$ la distance du chat c à la souris m suivant la notion de distance définie dans l'environnement.

$$\text{rew} = - \sum_{c \text{ in chats}} (d(c, m))^2 \quad (4.2)$$

Nous avons ici choisi de faire la somme des carrés afin de favoriser les mouvements où les chats se rapprochent tous et éviter qu'un chat puisse aller se ballader à l'opposé de sa destination.

Notons que, dans le cas de plusieurs souris, la méthode renverra le reward par rapport à la souris la plus avantageuse (et donc la plus proche).

4.3.3 Q-Learning

Le coeur du Q-Learning se trouve dans la méthode action de l'objet chat. En effet, nous allons effectuer une action et donc pouvoir observer le résultat de cette action sur l'environnement. De cette observation, on déduira le reward à associer avec l'action.

L'algorithme de Q-Learning utilisé (algorithme 4) est décrit dans [Littman 1994]. Il présuppose que nous avons reçu un reward rew après avoir effectué l'action a .

Algorithme 4: Algorithme de Q-Learning

- (1) $Q[s, a] := (1 - \alpha) * Q[s, a] + \alpha * (\text{rew} + \gamma * V[s'])$
- (2) $V[s] := \text{best } Q[s, .]$
- (3) for a in actions :
- (4) if a in best :
- (5) $\text{pi}[s, a] := 1/L$
- (6) else :
- (7) $\text{pi}[s, a] := 0$
- (8) $\alpha := \alpha * \text{decay}$

Rew est donc la valeur du reward dû à l'action a que l'on vient d'effectuer. $Alpha$ est le learning rate décrit dans l'équation 2.15. Ce learning rate va décroître au cours du temps suivant le paramètre $decay$.

$Gamma$ est bien entendu le facteur d'amortissement de l'équation 1.2.

Best $Q[s,.]$ désigne tout simplement la meilleure valeur de Q à l'état s . Le paramètre L est lui le nombre d'actions à l'état s dont la valeur $Q[s,a] = \text{best } Q[s,.]$. De cette manière, on voit que les probabilités sont équitablement réparties entre les meilleures possibilités.

Une autre solution, plus douce, consisterait à répartir les probabilités non pas entre les meilleures actions mais entre toutes les actions suivant leur valeur.

Algorithme 5: Algorithme de Q-Learning adouci

- (1) $Q[s,a] := (1-\alpha)*Q[s,a] + \alpha*(rew+\gamma*V[s'])$
- (2) $V[s] := \text{best } Q[s,.]$
- (3) for a in actions :
- (4) $pi[s,a] := Q[s,a]/\text{Total } Q$
- (5) $\alpha := \alpha*decay$

Avec Total Q défini comme la somme des valeurs de toutes les actions à l'état s .

$$\text{Total } Q = \sum_{a \text{ in actions}} Q[s, a] \quad (4.3)$$

Le problème de cette méthode est dans la gestion des valeurs $Q[s,a]$ négatives. Sa pertinence est aussi questionnable : est-ce qu'accorder une probabilité non-nulle aux solutions non-idéales ne fait pas double emploi avec l'exploration ?

C'est donc l'algorithme 4 qui est implémenté. Néanmoins, nul doute que l'algorithme 5 mériterait d'être testé.

4.3.4 Exploration/Exploitation

Comme nous l'avons vu au chapitre précédent, il existe plusieurs méthodes pour prendre en compte le dilemme exploration/exploitation. La solution retenue dans cette implémentation est celle décrite sous le nom *Exploration*

stochastique décroissante.

Pour ce faire, nous fixons donc un paramètre appelé *exploration_coeff*. Ce paramètre représente la probabilité de choisir une action aléatoirement plutôt que de choisir la meilleure solution connue.

Ce paramètre décroît au cours du temps suivant le paramètre *explore_decrease*.

L'action choisie a donc une probabilité *exploration_coeff* d'être complètement aléatoire. Sinon, dans le cas de l'exploitation, l'action est choisie a avec une probabilité $\pi[s,a]$.

$$\text{exploration_coeff}(t + 1) = \text{exploration_coeff}(t) * \text{explore_decrease} \quad (4.4)$$

L'utilisateur peut fixer ces deux valeurs à sa guise. La décroissance est amorcée lors de l'apprentissage et continuée lors du run en lui-même. Il n'y a donc pas de remise à zéro du coefficient d'exploration. Ce choix est volontaire, on suppose en effet qu'il est plus intéressant de disposer d'un grand coefficient d'exploration durant l'apprentissage et de favoriser l'exploitation lors du run. Cela peut cependant être facilement changé.

Notons que poser la constante *explore_decrease* à 0 revient à une exploration stochastique classique telle que décrite dans [Littman 1994].

À titre de perspective, il serait intéressant de comparer les performances de l'exploration stochastique décroissante avec celle d'une implémentation de la méthode de l'entropie décrite dans [Achbany et al, 2006].

4.4 La souris

La souris n'utilise quant à elle pas de technique de Reinforcement Learning. Pour choisir son action, la souris utilise donc l'algorithme Best Escape stochastique (algorithme 1). L'algorithme a cependant été légèrement modifié : la souris tente de minimiser la somme des carrés des distances entre elle-même et les chats et, si aucune solution n'est possible, la souris reste sur place.

Algorithme 6: Algorithme best-escape implémenté

- (1) $s = \text{etat}, d_{total} = 0$
- (2) available actions = actions for souris(s)
- (3) total_d := 0
- (4) for a in available actions :
- (5) $d = 0$
- (6) for c in chats :
- (7) $d = d + (d(c, mouse))^2$
- (8) $d(a) := d$
- (9) total_d := total_d + d
- (10) choose action a with probability $\frac{d(a)}{\text{total_d}}$

Le mouvement de la souris est donc bel et bien stochastique.

4.5 Déroulement du programme

Lorsqu'une simulation est lancée, il est fait appel à la méthode *main*. Cette méthode *main* comporte trois sections : initialisation, apprentissage, run.

L'initialisation consiste, on s'en doute, à placer chacun des éléments dans l'environnement et à créer tous les objets nécessaires.

Vient ensuite la période d'apprentissage : durant cette période, aucune visualisation n'est lancée. Le jeu va se dérouler durant un nombre déterminé de coups (paramètre *training* dans le fichier des constantes). Cette période, qui peut être réduite à 0, permet de remplir autant que possible les matrices Q , V et pi . Si le but est atteint lors de l'entraînement, les agents sont remis à leur position initiale dans l'environnement.

À titre d'information, une ligne indique chaque centaine de coups joués, permettant de suivre l'avancement de la période d'apprentissage.

La dernière partie est le run lui-même. Les agents sont remis à leur position initiale. Le jeu est ensuite joué, à raison d'un tour toutes les n secondes (n est le paramètre *to_sleep* dans le fichier des constantes). Le jeu se joue jusqu'à la victoire du chat, auquel cas le nombre de coups nécessaires est affiché, ou jusqu'à un nombre de coups maximal (paramètre *run* dans le fichier des constantes).

Remarquons que l'apprentissage est bien entendu entièrement facultatif. Il suffit de mettre le paramètre `training` à 0.

Des informations sur les matrices Q , V et pi sont affichées après l'apprentissage et après le run.

4.6 Visualisation

Il est toujours agréable de disposer d'une visualisation du problème étudié. Afin de rester le plus général possible, cette visualisation n'est pas automatique. Cependant, l'environnement dispose d'une méthode *display* qui peut être appelée à n'importe quel moment. Dans notre implémentation, la méthode *display* est appelée après chaque action dans la boucle principale du programme. De cette manière, on peut suivre étape par étape et en direct l'évolution du mouvement des agents.

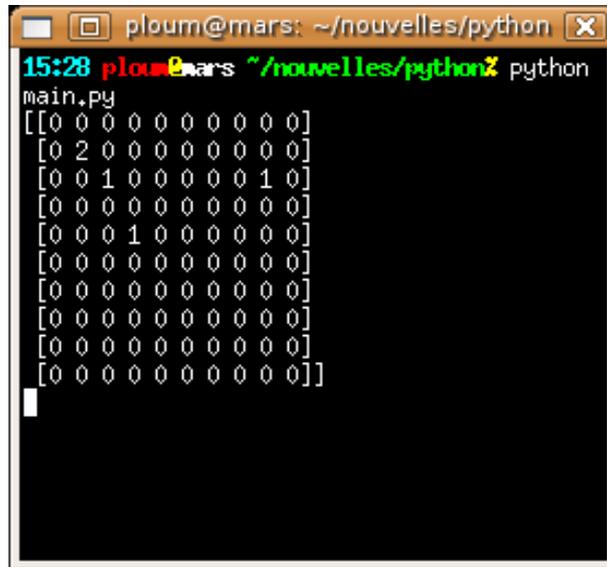
La méthode *display* ne renvoie par défaut qu'une simple matrice représentant l'environnement. Une première possibilité de visualisation consiste simplement à afficher cette matrice avec la commande `print` de Python. On obtient alors un résultat similaire à celui de la figure 4.4.

Il est bien sûr possible de passer la matrice à n'importe quelle méthode de visualisation, ce qui rend le framework facilement portable ou adaptable. À titre d'exemple, nous avons implémenté une visualisation utilisant la librairie *LiveWires* (voir [McCaughan 2001]). La librairie *LiveWires* est une librairie permettant de programmer facilement quelques jeux graphiques en agissant comme interface par dessus Tk.

Notons que la situation présentée à la figure 4.5 est identique à celle affichée dans la figure 4.4. La différence observée est simplement due au fait que l'origine du repère de la matrice est en haut à gauche alors que *LiveWires* utilise une origine plus traditionnelle (en bas à gauche).

Il est bien sûr tout à fait possible d'envisager n'importe quelle autre méthode de visualisation. On peut aussi imaginer de travailler dans un environnement à 3 dimensions visualisé via une méthode appropriée. Le résultat de la méthode *display* est en effet toujours une matrice aux dimensions du problème.

Enfin, remarquons que la visualisation n'est pas utilisée lors de l'appren-



```
ploum@mars: ~/nouvelles/python
15:28 ploum@mars ~/nouvelles/python python
main.py
[[0 0 0 0 0 0 0 0 0 0]
 [0 2 0 0 0 0 0 0 0 0]
 [0 0 1 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]]
```

FIG. 4.4 – Affichage de la matrice environnement en console par la méthode display

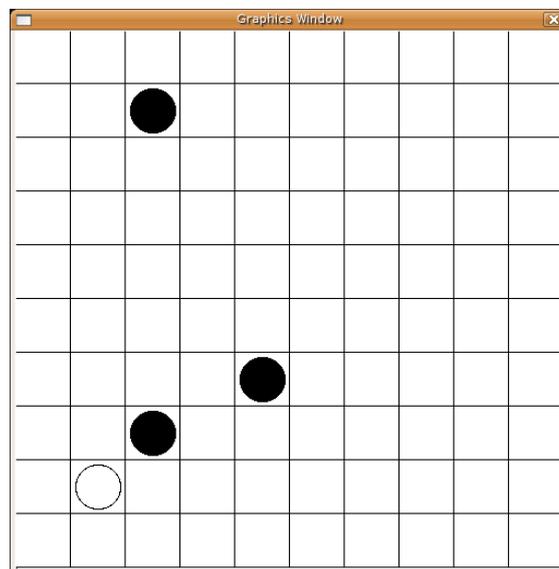


FIG. 4.5 – Affichage de l'environnement via la librairie LiveWires

tissage et ce pour des soucis de performances. La visualisation ralentit en effet considérablement le traitement. De plus, pour que la visualisation soit humainement intéressante, un délai doit être introduit entre chaque mouvement. Ce délai est la constante *to_sleep*, fixé à 0.1s par défaut mais modifiable par l'utilisateur dans le fichier des constantes.

4.7 En résumé

Ce chapitre a donc été la mise en commun des trois premiers chapitres. Les notions théoriques des chapitres 1 et 2 ont été mises en pratique et implémentées dans le cadre du problème décrit dans le chapitre 3.

Nous avons donc abordé de manière plus technique les différents éléments qui composent notre framework, à savoir l'environnement, les agents, les sub-agents ainsi que la notion d'objet permettant de construire un agent.

Des problèmes absolument imprévus se sont présentés (par exemple l'association d'une action de l'agent avec les mouvements des subagents) et ont été résolus.

Notons aussi que dans la section consacrée à l'identification d'un état, nous nous sommes rendus compte que l'optimisation pouvait permettre de percevoir un problème sous un jour nouveau et non seulement améliorer la performance brute mais aussi la performance du modèle en lui-même.

Chapitre 5

Simulations et expériences

5.1 Recherche classique dans l'arbre des solutions sans coopération

La première implémentation réalisée n'est en fait pas une implémentation de Reinforcement Learning au sens strict. Il s'agit plus d'une implémentation d'un min-max où la valeur d'un état serait le reward précédent. Cette implémentation a servi d'analyse préliminaire du problème.

Le principe en est relativement simple : la souris effectue un best escape non stochastique et les chats, quant à eux, explorent l'arbre des solutions depuis la position où ils sont actuellement. Malheureusement, on le comprendra aisément, cette exploration est limitée en profondeur par les performances de la machine. En pratique, il s'agit d'une profondeur 4 ou 5. Notons que le mouvements des chats est parfaitement déterministe et individualiste. Il n'y a ni stochasticité, ni coopération.

5.1.1 Le problème du parallélisme et la symétrie du problème

Cette expérience préliminaire a permis de mettre en évidence un problème pour le moins inattendu et passablement important : le parallélisme.

En effet, en l'absence de stochasticité, une direction primera nécessairement sur les autres dans l'implémentation, selon que le symbole de comparaison soit $<$ ou \leq . Or, rappelons que nous sommes dans un environnement utilisant la distance de Manhattan. Il s'en suit que si la direction prioritaire est le haut (par exemple, c'est d'ailleurs le cas dans l'implémentation), un

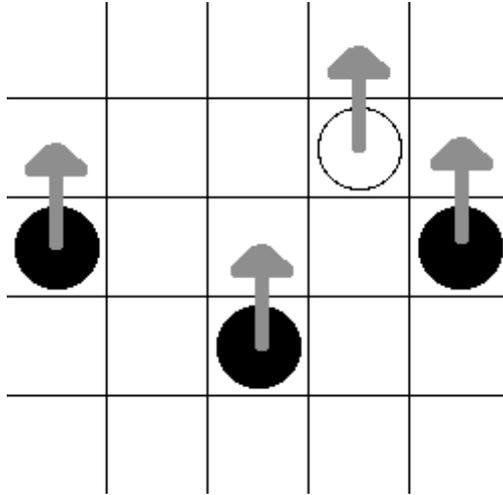


FIG. 5.1 – Illustration du problème du parallélisme

chat souhaitant attraper une souris ira toujours d'abord vers le haut¹ et se déplacera horizontalement uniquement lorsque la souris sera dans le même plan horizontal que lui. Or, une fois qu'il se déplacera suivant un plan horizontal, il prendra du retard par rapport à la souris, verticalement parlant. Depuis à peu près n'importe quelle situation initiale, on arrive donc très vite à un équilibre où, à chaque étape, la situation est tout simplement translatée d'une case vers la direction prioritaire. Le problème est illustré sur la figure 5.1

Ce problème rend en fait toute tentative d'attraper la souris sans coopération presque impossible sans un nombre conséquent de chats. C'est aussi une illustration particulièrement appropriée du fait qu'un raisonnement déterministe est parfois moins efficace qu'un raisonnement stochastique.

5.2 Mode opératoire

À moins que cela ne soit explicitement précisé, toutes les expériences qui suivent ont été réalisées en suivant le même mode opératoire. Un *run* consiste en une période d'apprentissage suivi d'une période d'exploitation

¹Il est évident que cela n'est que dans le cas où le haut est l'une des directions permettant d'attraper la souris en un minimum de mouvements. Si la souris se trouve juste au dessous de lui sur la droite, le chat utilisera la direction prioritaire parmi le bas et la droite. Toutes les directions sont en effet ordonnées.

pendant laquelle le chat tente d'attraper la souris. Cette dernière est appelée *concours* dans la suite du texte. Le nombre de coups qu'il met pour y parvenir est le *résultat* du run. Notons que l'apprentissage est réinitialisé et refait entièrement à chaque run.

Pour mesurer un facteur précis et donc pour faire une expérience, nous avons lancé entre 10 et 30 fois le même run, les plus lents n'étant bien sûr lancés qu'une dizaine de fois. Nous avons ensuite calculé la moyenne et l'écart type des résultats des runs. C'est cette moyenne qui est présentée comme résultat d'un test donné.

Notons aussi que les valeurs trop extrêmes et qui semblent non-reproductibles après plusieurs dizaines d'expériences sont écartées.

Par défaut, nous utiliserons les valeurs des paramètres suivantes :

- *mouse_sleepness*= 0.0
- *training* (nombre de coups d'apprentissage)= 10000
- *coefficient d'exploration (valeur de départ)*= 0.9
- *gamma*= 0.9
- *alpha*= 1.0
- *decay*= 0.9999

Afin d'avoir une base de comparaison, nous précéderons chaque expérience d'une série de runs d'un agent aléatoire (AA). Cet agent choisit donc parfaitement au hasard l'action à effectuer parmi les différentes possibilités. Il n'y a donc, bien sûr, aucun apprentissage.

Un agent entièrement déterministe sera, lui, confronté au problème du parallélisme précédemment évoqué. Tester avec un agent déterministe n'a donc foncièrement pas beaucoup d'intérêt : soit les conditions initiales sont, par chance, favorables, et la souris est attrapée en quelques coups, soit le programme boucle. Après quelques vérifications, il s'avère que la majorité des situations sont dans le second cas.

5.3 Expérience 3 chats et une souris

Comme nous l'avons démontré dans le chapitre 3, il est impossible pour deux chats d'attraper une souris. Nous avons donc choisi de mener une expérience comportant 3 chats et une souris dans un espace de 5x5.

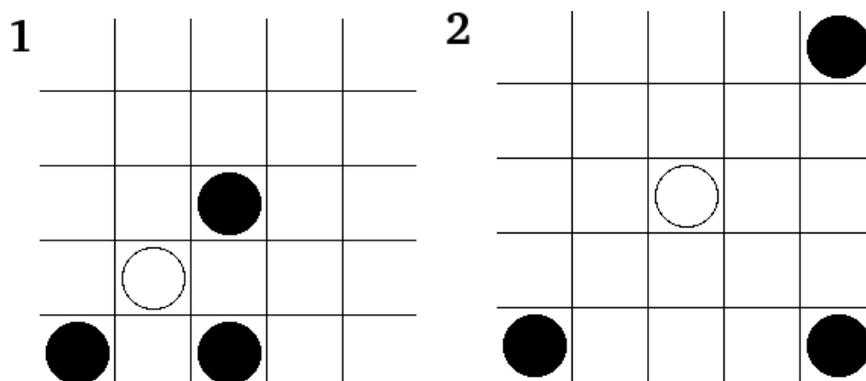


FIG. 5.2 – Les deux situations initiales testées avec 3 chats/1 souris

5.3.1 Situations de test

Nous avons testé deux configurations différentes. Les deux situations initiales sont visibles sur la figure 5.2.

La première situation possède la particularité d’être possible à résoudre en un seul coup. La seconde situation, au contraire, lance les chats aux points les plus éloignés possibles de la souris.

Remarquons que nous avons lancé pour ces deux situations un agent chat parfaitement aléatoire (AA) afin d’établir une base de comparaison non-biaisée. Il s’avère que, pour la situation 1, un agent aléatoire mettra en moyenne plus de 700 coups pour atteindre la solution avec des pointes à 1700-2000 coups et des minimas aux alentours de 400. Dans la situation 2, par contre, la moyenne aléatoire est plus proche de 1500 coups avec des pointes à 2000 ou 3000. Remarquons que dans de rares cas, un agent entièrement aléatoire mettra très peu de coups et pourra même battre le Reinforcement Learning lorsque celui-ci fait un mauvais score. Le Reinforcement Learning étant lui-même très dépendant du hasard, ce résultat n’est guère étonnant. Un résultat aléatoire n’ayant que des mauvais scores ne serait assurément pas parfaitement aléatoire.

5.3.2 Calcul du paramètre *explore_decrease*

Comme expliqué dans l’équation 3.5, nous avons introduit un paramètre de décroissance de l’exploration. Cependant, afin que les tests soient comparables, il faudrait que le paramètre soit adapté à la période d’apprentissage.

[Littman 1994] ayant fixé le coefficient d’exploration à 0,2, il nous a semblé judicieux de tenter d’obtenir un coefficient compris entre 0,1 et 0,2 après la période d’apprentissage. Nous avons donc arbitrairement fixé la valeur du coefficient d’exploration à 0,121, valeur qui paraît adaptée au moment de commencer le concours, une fois la période d’apprentissage terminée. De cette manière, quelle que soit la durée de l’apprentissage, toutes les expériences se retrouveront au même point avant de démarrer.

Nous avons donc calculé la valeur de D_{explor} en fonction de la taille de l’apprentissage n :

$$D_{explor} \simeq n \sqrt{\frac{0.121}{0.9}} \quad (5.1)$$

Bien sûr, la valeur obtenue à l’équation 5.1 est arrondie, c’est la raison du symbole \simeq . Étant donné le nombre d’expériences et la part de hasard inhérente au problème, il semble tout à fait justifié de procéder de cette manière. Ainsi, pour un apprentissage de 10.000 coups, on choisira $D_{explor} = 0.9998$. Pour 1000 coups, ce sera 0.998 et pour 5000 on obtiendra 0.9996.

5.3.3 Influence de l’apprentissage

La première expérience réalisée a été de faire varier le taux d’apprentissage. Par taux d’apprentissage, nous entendons le nombre d’actions effectuées durant la période d’apprentissage. Les résultats en fonction des taux d’apprentissages sont visibles pour les deux situations sur les figures 5.3 et 5.4. Ces graphiques ont été obtenus en faisant une moyenne du résultat sur 30 runs pour un taux d’apprentissage donné. Les taux d’apprentissage 50.000 et 100.000 ont été moyennés sur respectivement 10 et 3 runs pour des raisons évidentes de temps de calcul.

En posant AA, un agent aléatoire et LX, un agent ayant un taux d’apprentissage de X, nous obtenons pour les situations 1 et 2 les résultats du tableau ci-dessous. Entre parenthèses, nous avons ajouté la valeur de l’écart type sur les résultats. Par souci de concision, certaines colonnes ne sont pas affichées dans ce tableau.

	AA	L0	L5000	L10000	L50000
S1	738 (702)	703 (504)	585 (202)	191 (88)	1812 (964)
S2	1394 (831)	804 (394)	361 (244)	677 (217)	∞

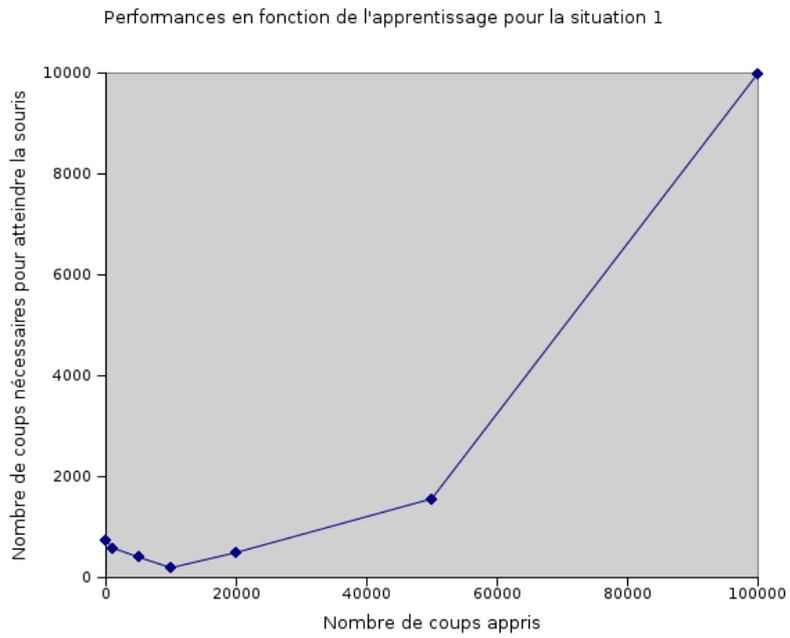


FIG. 5.3 – Influence de l'apprentissage dans la situation 1

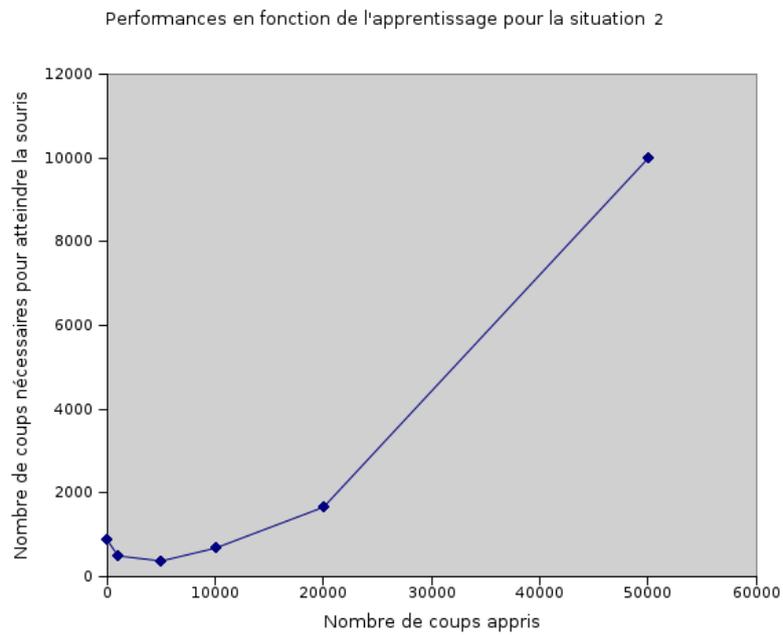


FIG. 5.4 – Influence de l'apprentissage dans la situation 2

La première chose qui frappe à la vision de ces graphiques est l'overfitting qu'une trop grande période d'apprentissage entraîne. Celui-ci est proprement catastrophique. Ainsi, au-dessus d'un certain seuil, les performances sont pires qu'un agent parfaitement aléatoire ! C'est à tel point que si on augmente encore la période d'apprentissage, le chat va tourner éternellement sans jamais attraper la souris. Il s'agit certainement d'un résultat que l'intuition n'avait pas prédit mais qui peut cependant s'expliquer : n'oublions pas que notre souris est stochastique ! Il est possible qu'un apprentissage trop poussé se concentre à apprendre des mouvements aléatoires. Si la souris ne reproduit plus exactement le même comportement, le chat devient donc incapable de s'adapter.

Notons aussi que les résultats sans apprentissage sont relativement bons bien que l'écart type soit pour cette série plus important (de l'ordre de 800) : c'est la grande force du Reinforcement Learning ! Il est capable d'apprendre en cours de route, sans apprentissage préalable. Cependant, dans ce cas, le hasard joue un grand rôle.

Les phases d'apprentissage ont été analysées et il s'avère que le taux d'exploration moyen d'un noeud (rappelons que notre agent peut faire ici 125 actions différentes, le taux d'exploration étant le nombre d'actions testées à ce noeud particulier) est extraordinairement constant : 17 ou 18 dans le cas de 10.000 mouvements d'apprentissages. De même, le nombre de fois que la souris a été attrapée durant l'apprentissage est aussi extraordinairement constant et fonction de la taille de l'apprentissage.

Enfin, un fait étonnant est apparu durant à peu près toutes les expériences réalisées : la dispersion des résultats. L'écart type est toujours assez important mais les résultats n'y sont pas observés de manière uniforme : on a une grosse concentration de très bons résultats et une autre concentration de très mauvais résultats. Cette dispersion n'est pas du tout observée dans les runs aléatoires.

Par exemple, pour la situation 1 avec 10.000 coups d'apprentissage, on observe 40% des résultats en-dessous de 25 coups, 20% en dessous de 100 coups et les derniers 40% compris entre 300 et 400 coups. De même, pour la situation 2 avec 5000 coups d'apprentissage, on observe : 30% en dessous de 50 coups, 20% en dessous de 300 et les 50% restant compris entre 400 et 800. Ces pourcentages ont été observés et reproduits plusieurs fois en lançant à chaque fois 10 runs d'un même test et sont visibles sur la figure 5.5. L'histogramme en gris foncé concerne la situation 1, celui en pointillés la situation

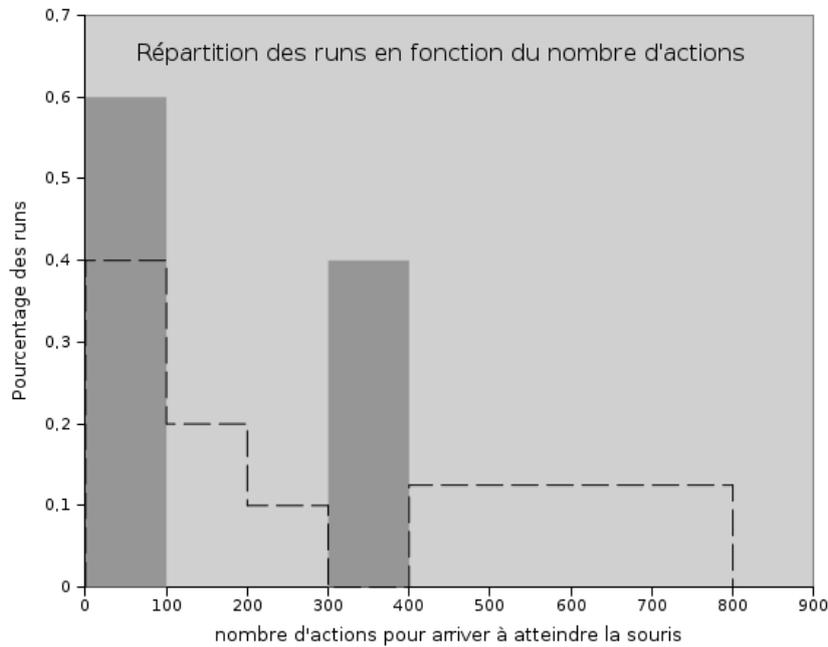


FIG. 5.5 – Répartition des runs en fonction du nombre d’actions pour atteindre la souris. La situation 1 est en gris foncé, la situation 2 en pointillés.

2. On voit très nettement qu’il y a un “trou” (bien que celui soit différent pour les deux situations) et que les résultats sont soit bons, soit mauvais mais sans réel intermédiaire.

De cette expérience, nous pouvons donc déduire deux choses très importantes. Premièrement, il est primordial de ne pas trop pousser l’apprentissage. En pratique, nous avons donc choisi la valeur de 10.000 (dans la situation 2, la moyenne donne l’avantage à 5000 mais en pratique 10.000 se révèle aussi une bonne valeur et moins “sensible” que 5000, dont les performances varient plus d’un problème à l’autre).

Second point : il semblerait que tous les apprentissages ne soient pas équivalents. Les résultats étant soit très bons, soit très mauvais, on peut en déduire que les chats se “perdent” si l’apprentissage ne possède pas certaines propriétés que nous n’avons pas pu déterminer. Cette “qualité d’apprentissage” n’est en tout cas pas liée au taux d’exploration par noeud ni au nombre de fois que la souris a été attrapée.

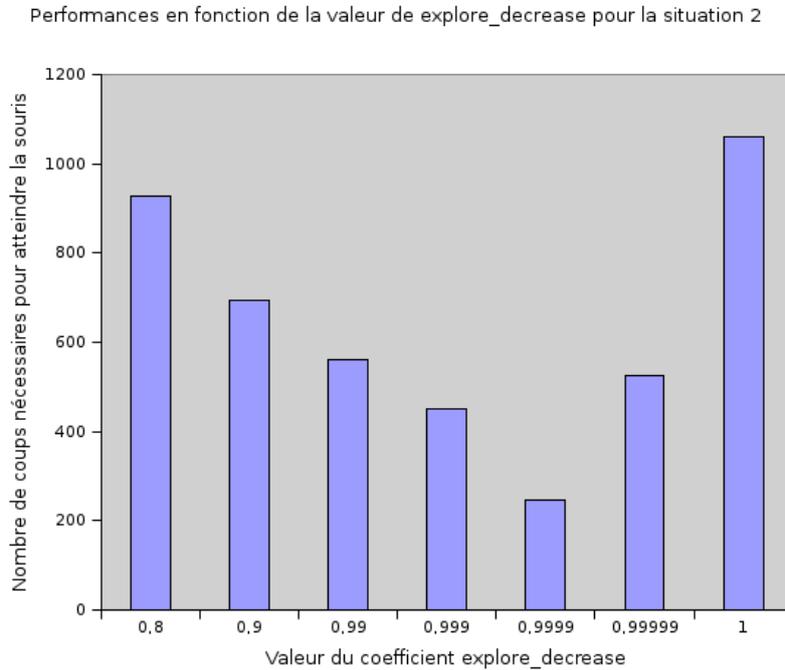


FIG. 5.6 – Influence du paramètre *explore_decrease* dans la situation 2

5.3.4 Influence du coefficient d’exploration

Avec l’équation 5.1, nous avons fixé arbitrairement la valeur du coefficient *explore_decrease*. Il nous a semblé utile d’en faire une analyse un peu plus poussée. Nous avons donc essayé plusieurs valeurs de ce coefficient sur la situation 2 avec 10.000 coups d’apprentissage. Le résultat est visible sur la figure 5.6.

Nous nous sommes alors aperçu que la valeur optimale de ce paramètre se situait à la valeur de 0.9999. De manière étonnante, cette valeur s’est révélée la plus efficace pour les autres essais, améliorant légèrement les performances et repoussant l’overfitting. C’est parfaitement logique car une valeur fixe rend les derniers apprentissages d’une grande série moins importants et donc moins propices à causer de l’overfitting.

A contrario, lorsque l’apprentissage est insuffisant, la valeur du coefficient d’apprentissage reste élevée lors du début du concours et cela permet d’apprendre en cours de route.

Remise à zéro du coefficient d'exploration avant le concours

Une autre idée que nous voulions tester était de mettre automatiquement le paramètre d'exploration à 0 lors du début du concours, en supposant que l'apprentissage a été suffisant. Cependant, cette pratique s'est révélée très dangereuse : beaucoup de runs se perdaient alors à 10.000 coups (limite à laquelle le run est arrêté même si la souris n'est pas arrêtée). Ceux qui réussissaient permettaient, il est vrai, une très infime amélioration. Cependant, la moyenne étant évidemment plombée par les 10.000+, l'idée a été abandonnée.

C'est, après réflexion, assez logique et important à être noté : le coefficient d'exploration ne doit jamais être nul. Une idée serait d'imposer une valeur minimale au coefficient d'exploration et de tester son effet sur l'overfitting après un grand nombre d'apprentissages.

5.3.5 Analyse de l'expérience

De cette expérience, nous avons pu tirer deux conclusions fondamentales pour la suite : notre programme est tout à fait fonctionnel mais, en contrepartie, le problème avec 3 chats est trop compliqué en raison de son branching factor. En effet, malgré les optimisations, le nombre d'états possibles est :

$$C_{24}^3 = 2024 \quad (5.2)$$

Comme chaque état permet 125 actions, cela nous donne une matrice $Q(s,a)$ d'une taille de 253000 entrées ! Remarquons aussi que le taux d'exploration maximum reste très faible : ainsi, même après 50.000 coups d'apprentissages, on ne dépasse pas 70 actions pour un noeud !

Ceci explique certainement les écarts-types élevés rencontrés.

5.4 Expérience 2 chats et une souris fatiguée

5.4.1 Conditions de l'expérience

La seconde approche est résolument plus simple : 2 chats et une souris dans la position visible à la figure 5.7. Le nombre d'états possibles est donc réduit :

$$C_{24}^2 = 276 \quad (5.3)$$

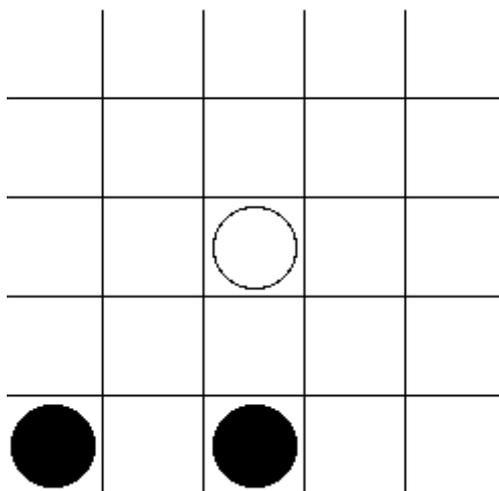


FIG. 5.7 – La troisième situation de nos expériences : 2 chats et une souris

Ce qui donne une matrice $Q(s,a)$ beaucoup plus raisonnable, à raison de 25 actions par état : 6900 entrées.

Pour cette expérience, le paramètre *mouse_sleepness* a donc été posé à la valeur arbitraire de 0.1. Le paramètre *explore_decrease* a, comme vu dans la section précédente, été posé à 0.9999.

Notons que dans la position décrite à la figure 5.7, un agent entièrement aléatoire attrape la souris entre 200 et 400 coups (moyenne de 240) même si on peut observer de rares très bons coups (entre 30 et 80). La fatigue de la souris est ici un facteur qui facilite énormément le travail d'un agent aléatoire.

5.4.2 Influence de l'apprentissage

Les résultats de l'expérience sont visibles sur la figure 5.8. Sans surprise, le problème est nettement mieux géré que le précédent. Notons qu'apparaît exactement la même courbe d'overfitting. Le minimum se produit ici après 1000 coups appris seulement. À 5000 coups, on est toujours en-dessous de l'aléatoire mais après 10.000, c'est l'explosion.

On notera cependant qu'à 10.000 coups d'apprentissage, l'agent est encore capable de très belles performances : 20 ou 30 coups. Celles-ci sont cependant alternées avec de terriblement mauvais résultats de l'ordre de 1000-1200

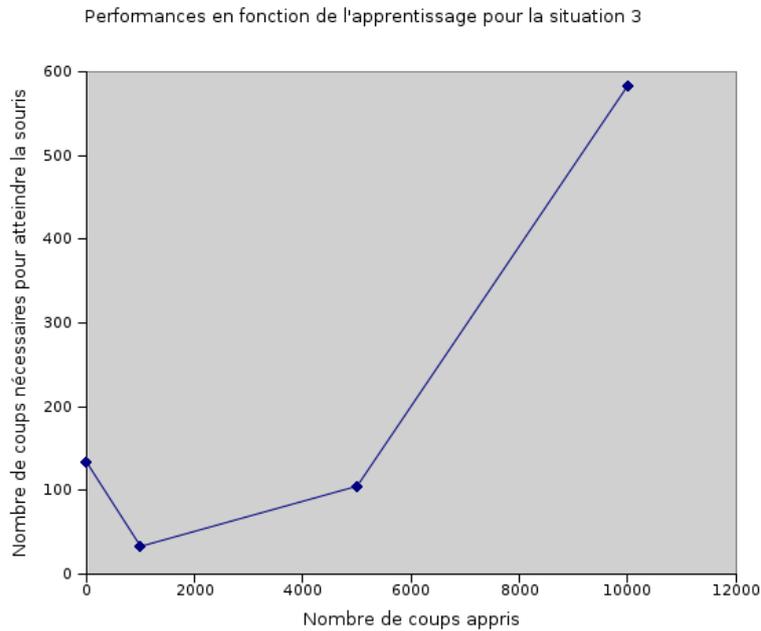


FIG. 5.8 – Influence de l'apprentissage dans la situation 3

coups.

Enfin, au niveau du taux d'exploration des noeuds, il est très bon : de l'ordre de 12-13 pour 1000 coups d'apprentissage (50%), 19-20 pour 5000 coups et 24 pour 10.000 coups.

5.5 Conclusions de l'expérimentation

Réaliser ce genre d'expériences prend beaucoup de temps et chaque observation doit être réalisée au minimum plusieurs dizaines de fois. Nous avons donc dû limiter le champ des observations mais le problème comporte un grand nombre de paramètres dont il serait intéressant d'étudier l'influence et les éventuelles corrélations.

Nous avons préféré nous centrer sur l'apprentissage qui nous semble primordial dans ce genre de problème. Il est important de constater que dans les 3 problèmes étudiés, les 3 courbes de résultats des figures 5.3, 5.4 et 5.8 ont une allure parfaitement similaire.

On en déduit que l'overfitting est un risque réel et particulièrement pernicieux. Mieux vaut apprendre moins que trop, quitte à réaliser l'apprentissage manquant dans le début du concours. Les bonnes performances réalisées sans le moindre apprentissage démontrent que l'apprentissage réalisé en cours de route est tout à fait utile : on ne refait pas les erreurs du passé.

Enfin, la grande disparité des résultats au sein d'une même série de runs fait soupçonner l'existence d'une certaine "qualité d'apprentissage".

À l'observation, il apparaît que les chats semblent toujours rester groupés ou du moins à une distance fixe l'un de l'autre. Cette méthode est particulièrement efficace pour encercler la souris lorsque celle-ci est tout près. Par contre, cette "coopération" se révèle inefficace si la souris s'éloigne un peu trop. En effet, les chats favorisent alors une coopération qui n'a pas lieu d'être : ils devraient se diriger le plus près possible de la souris sans se préoccuper de l'autre chat. C'est peut-être là que réside une explication des "runs perdus", runs durant lesquels le chat met un temps fou pour attraper la souris.

En conclusion, on peut dire que les résultats sont peu satisfaisants bien qu'ils prouvent que la méthode est fonctionnelle. Les deux points critiques de la méthode et qui pourront être sujet à amélioration sont :

1. L'overfitting, qui devra être mieux compris en fonction de l'influence des autres paramètres (α, γ) .
2. Le nombre d'états, qui pourra être réduit en effectuant, par exemple, un lissage sur les états similaires.

Ces deux points pourraient faire l'objet d'un travail ultérieur.

Conclusion et perspectives

Lors de l'introduction de ce mémoire, nous nous étions fixé deux objectifs : l'implémentation d'un framework de Reinforcement Learning et l'étude de l'éventuelle coopération entre agents suivant la méthode "agent/sous-agents".

Avant de nous attaquer au problème proprement dit, nous avons dû passer en revue toute la théorie du Reinforcement Learning et des jeux de Markov. Mais le problème s'est révélé bien plus complexe qu'entrevu au départ. Qui aurait pu croire qu'un problème à l'énoncé si simple, des chats cherchant à attraper une souris, pouvait receler tant de complexité, tant de possibilités et de difficultés techniques ?

Ce mémoire, loin d'être uniquement théorique, a donc nécessité des pans entiers purement techniques pour l'implémentation et la résolution de tous ces petits problèmes imprévus et pourtant fondamentaux. On peut donc affirmer que le premier objectif a été rempli.

Enfin, ce mémoire a montré que sans coopération, il est presque impossible pour des chats d'attraper la souris, problème de parallélisme évoqué au chapitre 5. Des chats non-coopératifs ne doivent donc, selon leur propre algorithme, que compter sur la chance pour attraper la souris. Or, les chats de ce mémoire ont réussi, sans discussion possible, des performances nettement meilleures que des chats aléatoires. D'ailleurs, bien que rendu difficilement interprétable à cause de la toricité de l'espace, le mouvement observé des chats ne semble pas du tout aléatoire. On croit plutôt voir une pince qui tente de se refermer sur une bulle de savon, bulle qui glisse et s'échappe à chaque fois de justesse. Il semble donc y avoir de la coopération mais l'enthousiasme scientifique est peut-être le seul responsable de cette interprétation.

Peut-être justement la coopération entre les chats est-elle trop forte ? En agissant comme un seul et unique agent, les chats sont incapables de calculer le bien-fondé d'une action individuelle. Ainsi, si un chat se rapproche de la

souris mais que le second chat force au même moment dans un obstacle, la résultante sera franchement négative. C'est, nous l'avons expliqué, le principal défaut de la méthode agent/sous-agents. Ajoutons à cela que le nombre d'actions et donc le branching-factor sont très élevés avec cette technique, expliquant les faibles performances obtenues pour 3 chats. Il serait donc particulièrement intéressant d'étudier les possibilités de coupler la technique agent/sous-agents avec des rewards individuels pour chaque sous-agent.

Perspectives

Comme nous l'avons vu tout au long de ce mémoire, le problème chat-souris est bien plus complexe qu'il n'y paraît à première vue. Ce mémoire n'a donc fait que débroussailler le terrain afin de permettre des études plus poussées. On peut ainsi imaginer des problèmes où les chats doivent explorer eux-mêmes l'environnement [Isler et al, 2004].

Il serait particulièrement intéressant de préciser la notion de "qualité d'apprentissage" et de pouvoir prédire les conditions qui rendent un run efficace ou non. De même, nous n'avons pu déterminer les conditions exactes qui introduisent un overfitting. Pouvoir prédire et quantifier l'overfitting serait assurément une grande avancée dans cette méthode.

Cependant, la perspective la plus enthousiasmante est sans conteste l'étude et l'amélioration de la coopération entre les sous-agents. À ce sujet, il faut noter l'approche Bayésienne proposée dans [Chalkiadakis & Boutilier, 2003]. Il serait particulièrement intéressant de comparer les résultats de cette méthode avec celle agent/sous-agents voire d'étudier la possibilité d'une méthode mixte qui combinerait les avantages des deux méthodes.

Un autre perfectionnement possible, qui pourrait peut-être conduire à une amélioration des résultats, est le lissage des états similaires. En effet, dans ce travail, nous avons considéré chaque état comme unique et indépendant des autres états. Or, il est plus que probable qu'un état aura tendance à avoir des valeurs $Q(s, \cdot)$ et $V(s)$ proches de celles des états similaires. C'est donc une voie à creuser qui pourrait éventuellement faire l'objet d'un travail ultérieur.

Le problème avait pourtant l'air si simple. Malgré cela, les idées nouvelles à lui appliquer continuent de pleuvoir. En Reinforcement Learning, ce ne sont certainement pas les perspectives qui manquent.

Bibliographie

- [Sutton & Barto, 1998] Richard S. Sutton & Andrew G. Barto, *Reinforcement Learning : An Introduction*, MIT Press, Cambridge, MA, 1998
- [Russel & Norvig, 2003] Stuart Russel & Peter Norvig, *Artificial Intelligence, A Modern Approach, second edition*, Prentice Hall, 2003
- [Tijms 2003] Henk C. Tijms, *A First Course in Stochastic Models*, Wiley, 2003
- [Martelli 2004] Alex Martelli, *Python en Concentré*, O'Reilly, 2004
- [Pilgrim 2004] Mark Pilgrim, *Dive Into Python*, <http://www.diveintopython.org/>, 2004
- [McCaughan 2001] Gareth McCaughan, *The LiveWires Python Course*, <http://www.livewires.org.uk/python/>, 2001
- [Nowé et al, 2005] Ann Nowé, Katja Verbeeck & Karl Tuyls, Learning Automata as a Basis for Multi-agent Reinforcement Learning, 16th European Conference on Machine Learning, Porto, Portugal, 2005
- [Achbany et al, 2006] Youssef Achbany, François Fouss, Luh Yen, Alain Pirotte & Marco Saerens, *Managing the Exploration/Exploitation Trade-Off in Reinforcement Learning*, submitted for publication, 2006
- [Littman 1994] Michael L. Littman, *Markov games as a framework for multi-agent reinforcement learning*, In Proceedings of the 11th International Conference on Machine Learning (ML-94), 1994
- [Littman 2000] Michael L. Littman, *Value-function reinforcement learning in Markov games*, Journal of Cognitive Systems Research, Vol. 2, No. 1, 2000
- [Hespanha et al, 1999] João P. Hespanha, Hyoun Jin Kim & Shankar Sastry, *Multiple-Agent Probabilistic Pursuit-Evasion Games*, Technical report, Dept. Electrical Eng. & Comp. Science, University of California at Berkeley, 1999
- [Isler et al, 2004] Volkan Isler, Sampath Kannan & Sanjeev Khanna, *Randomized Pursuit-Evasion with Limited Visibility*, In Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA), 2004

- [Chalkiadakis & Boutilier, 2003] Georgios Chalkiadakis & Craig Boutilier, *Coordination in Multiagent Reinforcement Learning : A Bayesian Approach*, In Proceedings of AAMAS'03, 2003
- [Könönen 2004] Ville Könönen, *Asymmetric multiagent reinforcement learning*, IEEE/WIC International Conference on Intelligent Agent Technology, 2004
- [Thrun 1992] Sebastian B. Thrun, *Efficient Exploration In Reinforcement Learning*, Technical Report CMU-CS-92-102, School of Computer Science, Carnegie Mellon University, 1992
- [Ratitch & Precup, 2003] Bohdana Ratitch & Doina Precup, *Using MDP Characteristics to Guide Exploration in Reinforcement Learning*, In Proceedings of Conference on Machine Learning, Cavtat-Dubrovnik, Croatia, 2003

Annexe A : Utilisation du programme

Lancement d'un run

Lorsqu'on effectue un run, le seul fichier nécessitant d'être modifié est le fichier `values.py`. Celui-ci contient tous les paramètres du problème. L'explication de chaque paramètre est indiquée en commentaire dans le fichier si nécessaire.

Notons que la configuration du problème lui-même est réalisée via 4 paramètres : *size*, *cats_pos*, *mouse_pos* et *stones_pos*.

Size est, on s'en doute, la dimension de notre espace. Il doit s'agir d'un tuple contenant deux valeurs entières. Les autres paramètres sont quant à eux des tuples contenant des tuples contenant les position de chaque sous-agent.

Ainsi, pour placer deux chats sur le tableau, on attribuera à *cats_pos* la valeur `[[4, 1], [2, 1]]`. On fait de même avec la souris et les obstacles (*stones*). Notons que la souris, elle, ne peut comporter qu'un et un seul tuple et s'écrit donc suivant la forme `[[2, 2]]`. On peut vouloir ne placer aucun obstacle, auquel cas la valeur de *stones_pos* sera tout simplement `[]`.

À titre d'exemple, remarquons qu'il est possible de recréer un espace entièrement fermé en posant des obstacles sur la ligne `[:, 0]` et sur la colonne `[0, *]` (voir le code source de `values.py`). Cependant, remarquons que les chats percevront toujours un espace torique et devront apprendre que passer à travers les murs n'est pas possible (le logiciel donne un fort reward négative pour toute tentative de ce type).

Une fois les paramètres voulus déterminés, il suffit de lancer le programme

avec la commande :

```
python main.py
```

Le déroulement de l'apprentissage va alors s'afficher : une ligne toutes les centaines de coups appris. Notons aussi que le mot "GOAL" apparaît chaque fois que le chat attrape la souris lors de l'apprentissage. Des informations sur les matrices Q , V et π du Q-Learning sont aussi affichées.

Après l'apprentissage, le concours se lance automatiquement. À la différence de l'apprentissage, chaque coup du concours est affiché et il est possible de suivre en direct l'évolution du jeu.

À la fin du jeu, le nombre de coups nécessaires (le résultat) est affiché. Le résultat est aussi automatiquement placé dans le fichier `results.txt` sous la forme :

```
a (b) [c]
```

Où a est le nombre de coups joués (le résultat), b est le nombre de fois que le but a été atteint durant l'apprentissage et c est le taux d'exploration d'un noeud maximum.

La boucle principale du programme se trouve dans le fichier `main.py` et il est évidemment possible de le modifier légèrement, par exemple pour commenter les lignes qui concernent l'affichage lors du lancement d'une série de simulations successives.

Implémenter un agent personnalisé

Le fichier `objets.py` contient la définition des agents. En fait, comme nous l'avons vu au chapitre 4, il ne s'agit pas tout à fait des agents mais des *objets* qui permettent de définir un agent.

Vous pouvez tout à fait définir votre propre objet qui doit implémenter les méthodes décrites à la figure 4.2, à savoir :

- `__init__(pos)` : le constructeur de l'objet qui prend en paramètre le tuple des tuples de positions des sous-agents.
- `nbr_actions()` : méthode qui ne prend pas de paramètre et renvoie un entier, le nombre d'actions possible de l'agent (ce nombre doit être

- valable pour chaque état)
- `actions(i, agents, env)` : méthode qui effectue l'action *i*. Sont aussi passés en paramètre *agents*, le tuple des tuples des positions actuelles des sous-agents et *env*, l'objet représentant l'environnement. La valeur retournée par la méthode doit être le nom de l'agent qui se trouve sur la case cible, que l'action ait échoué ou non (la case cible est généralement vide).
- `find_best_move(env, agents)` : renvoie l'action choisie pour des sous-agents aux positions *agents* dans l'environnement *env*. Cette méthode doit prendre en compte une éventuelle exploration.

Remarquons que la difficulté a été autant que faire se peut déplacée dans le programme lui-même. L'utilisateur souhaitant implémenter son propre agent n'a donc pas à maintenir les états des ses sous-agents ni de l'environnement global.

Enfin, une fois l'agent implémenté, il est nécessaire de le créer et l'ajouter dans l'espace dans le fichier `main.py`. Ainsi, pour un agent intitulé `obj_chat`, on aura le code suivant :

```
c_position = valeurs.cats_pos
objchat = obj_chat(c_position)
chats = espace.addagent(objchat, c_position)
```

Ensuite, dans le corps de la boucle, que ce soit pour l'apprentissage ou pour le concours, il est bien sûr nécessaire de faire agir l'agent :

```
result = chats.best_move()
```

En vérifiant que le tuple de valeurs *result* ne contienne pas l'objectif désiré (ici un 2 qui représenterait la souris capturée).

Affichage personnalisé

L'affichage de l'environnement est entièrement décrit dans la méthode *show* du fichier `display.py`. Cette méthode reçoit en paramètre une matrice décrivant l'environnement. Il est donc très simple de programmer sa fonction d'affichage personnalisée.

```
def show(matrice) :  
    print matrice  
    time.sleep(valeurs.to_sleep)
```

Remarquons que le résultat de la figure 4.4 peut être obtenu tout simplement en implémentant la méthode `show` de la façon décrite ci-dessus.

Annexe B : Codes sources

Le code source fournit en annexe comporte les fichiers suivants :

- `main.py` : Le fichier principal dans lequel se trouve la boucle du programme.
- `values.py` : les fichiers contenant les paramètres.
- `objets.py` : Les différentes implémentations des objets permettant de construire les agents.
- `m1.py` : Implémentation des agents et sous-agents. Cette implémentation utilise les méthodes de *objets.py*.
- `m1_envir.py` : Implémentation de l'environnement et de ses composantes (distance, reconnaissance d'état).
- `display.py` : Implémentation d'une méthode d'affichage utilisant *LiveWires*. Cette méthode a été utilisée pour les illustrations de ce mémoire.
- `jeu1.py` : Implémentation de la méthode sans coopération expliquée dans la section 5.1.

Le code source de la librairie graphique *LiveWires* n'est pas inclus. Il est disponible à la référence [McCaughan 2001].

main.py

```
1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3 import random
4 from ml import *
5 from ml_envir import *
6 from objets import *
7 import display
8 import values
9 import os
10
11 global goal
12
13 if __name__ == "__main__":
14     global valeurs
15     global players
16     goal = 0
17
18     valeurs = values.parameters()
19     size = valeurs.size
20     ##### MODIFY HERE #####
21     #Cats
22     c_position = valeurs.cats_pos
23     #Mouse : adding multiples mouse can have unpredictable results
24     s_position = valeurs.mouse_pos
25     #Stones (cannot be moved)
26     stone_pos= valeurs.stones_pos
27     #####
28     begin_graphics(width=valeurs.size_x , height=valeurs.size_y)
29
30
31     #matrice =[[0,0,0,0,0,0,0,0,0,0],
32     #         [0,2,0,0,0,0,1,0,0,0],
33     #         [0,0,0,0,0,0,0,0,0,0],
34     #         [0,0,0,0,0,0,0,0,0,0],
35     #         [0,0,0,0,0,0,0,0,0,0],
36     #         [0,0,0,0,0,0,0,0,0,0],
37     #         [0,0,0,0,0,0,0,0,0,0],
38     #         [0,0,0,0,0,0,0,0,0,0],
39     #         [0,0,0,0,1,0,0,0,0,0],
40     #         [0,0,0,0,0,0,0,0,0,0]]
41     #show(matrice)
42
```

```

43  #sleep(1000)
44
45
46
47
48  objchat = obj_chat(c_position)
49  objsouris = obj_souris(s_position)
50  objstone = obj_stone(stone_pos)
51
52
53  envi = obj_env(size)
54  espace = enviro(envi)
55  chats = espace.addagent(objchat, c_position)
56  souris = espace.addagent(objsouris, s_position)
57  stones = espace.addagent(objstone, stone_pos)
58  display.show(espace.display())
59
60  count = 0
61  training = valeurs.training
62  run = valeurs.run
63  cent = 0
64  centaine = 0
65  learn_goal = 0
66  while count < training :
67      result = chats.best_move()
68      if result.count(2) >= 1 :
69          print "reinitialize the space"
70          espace.reinit()
71          learn_goal += 1
72      r = random.random()
73      if r >= valeurs.mouse_sleepness :
74          souris.best_move()
75      count += 1
76      cent +=1
77      if cent >= 100 :
78          centaine += 1
79          cent = 0
80          print "%d mouvements ont ete appris"%(100*centaine)
81
82      #display.show(espace.display())
83
84
85  print "on a atteind %d fois le goal durant l'apprentissage"%learn_goal

```

```

86     b = learn_goal
87     print "##### Q #####"
88     c = espace.q_infos()
89     print "##### V #####"
90     espace.v_infos()
91     print "##### pi #####"
92     espace.pi_infos()
93
94     espace.reinit()
95     #print espace.display()
96     display.show(espace.display())
97     #time.sleep(1000)
98
99     #valeurs.exploration_coef = 1.0
100    count = 0
101    cent = 0
102    centaine = 0
103    while chats.best_move().count(2) < 1 :
104        r = random.random()
105        if r >= valeurs.mouse_sleepness :
106            souris.best_move()
107        count += 1
108        #print "Mouvement num %d"%count
109        display.show(espace.display())
110        #print espace.display()
111        if count > run :
112            break
113        if cent >= 100 :
114            centaine += 1
115            cent = 0
116            print "%d mouvements ont ete realises"%(100*centaine)
117    print "on a fait %d mouvements"%count
118    a = count
119    print "\n\nApres le run\n\n"
120    print "##### Q #####"
121    espace.q_infos()
122    print "##### V #####"
123    espace.v_infos()
124    print "##### pi #####"
125    espace.pi_infos()
126
127    abc = "%d (%d) [%d]\n"%(a,b,c)
128    fd=open("result.txt",mode ='a+')

```

```
129     fd.write(abc)
130     fd.close()
131
132     #chats.do_action(5)
133     #espace.display()
134     #souris.best_move()
135     #souris.best_move()
136     #souris.best_move()
137     #espace.display()
```

values.py

```
1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3
4 class parameters :
5     def __init__(self):
6         #simulation
7         self.mouse_sleepness = 0.1
8         #nombre de coups pour l'entrainement
9         self.training = 20000
10        #nombre de coups maximum que peut effectuer le chat lors du concours
11        #avant d'etre declare comme perdant.
12        self.run = 10000
13
14
15        #constantes
16        #####
17        # exploration
18        self.exploration_coeff = 0.9
19        #self.exploration_coeff = 1.0
20        #avec une valeur de 0.998, le coeff d'exploration retombe a 0.121 a
21        #avec une valeur de 0.9998 on est a 0.121 apres 10.000 coups
22        # 0.9996 donne 0.121 apres 5000 coups
23        # 0.99996 donne 0.121 apres 50000 coups
24        # 0.9999 pour 20000
25        self.explore_decrease = 0.9996
26        #self.explore_decrease = 1.0
27        #amortissement de l'expected return
28        self.gamma = 0.99
29        #learning rate
30        self.alpha = 1.0
31        #decay of the learning rate
32        #self.decay = self.explore_decrease*self.explore_decrease
33        self.decay = 0.9999
34
35        #parametres du probleme
36        #####
37        #les deux dimensions
38        self.size = [10,10]
39        #self.size = [5,5]
40
41
42        # Le probleme est prevu initialement pour
```

```

43     # au moins un chat et une souris
44     # Changer le nombre requierent d'implementer en fonction
45     # les objets agent dans main.py
46
47     # un tableau contenant les coordonnees des chats
48     # Les chats doivent etre deux ou plus (voire tout seul)
49     self.cats_pos = [[4,1],[2,1]]
50     #self.cats_pos = [[0,3],[2,1],[4,2]]
51     #self.cats_pos = [[0,0],[4,0],[4,4]]
52     #self.cats_pos = [[4,3],[2,8]]
53     #self.cats_pos = [[4,3],[2,8],[2,2]]
54     # un tableau contenant les coordonnees des souris
55     # La souris doit etre seul
56     self.mouse_pos = [[2,2]]
57     # Les obstactles. Le tableau peut etre vide ou contenir un nombre
58     # indetermine de pierres.
59     #self.stones_pos = [[4,6]]
60     self.stones_pos = []
61     #self.stones_pos = [[0,0],[0,1],[0,2],[0,3],[0,4],[0,5],[0,6],[0,7]
62     [0,0],[1,0],[2,0],[3,0],[4,0],[5,0],[6,0],[7,0]]
63
64     #Constantes d'affichage
65     #####
66     self.k = 50
67     self.size_x = self.k*self.size[0]
68     self.size_y = self.k*self.size[1]
69     #temps a attendre entre chaque coup.
70     self.to_sleep = 0.1

```

objets.py

```
1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3 import random
4 from ml import *
5 from ml_envir import *
6 import values
7
8
9 class obj_stone :
10     def __init__(self, pos) :
11         global valeurs
12         valeurs = values.parameters()
13         self.name = 3
14     def nbr_actions(self) :
15         return 0
16     def actions(self, i, agents, env):
17         print "I'm a stone"
18     def find_best_move(self, env, agents) :
19         return 0
20
21 #####
22
23 class obj_chat :
24     def __init__(self, pos) :
25         global valeurs
26         valeurs = values.parameters()
27         self.name = 1
28         self.nbr = len(pos)
29         #nbr mouvement par subagent
30         self.base = 5
31
32     def nbr_actions(self):
33         n = self.base**self.nbr
34         return n
35
36     def action(self, i, agents, env) :
37         previous_state = env.get_state_name()
38         depl=[]
39         level = 0
40         mouve = i
41         while level < self.nbr :
42             zemv = mouve%self.base
```

```

43         depl.append(agents[level].move(zenv))
44         mouve = mouve/self.base
45         level += 1
46     # Heart of the Q-Learning methode
47     current_state = env.get_state_name()
48     #reward
49     rew = self.reward(depl, agents, env)
50     #We set the new Q
51     q_prime = env.get_q_value(current_state, i)
52     v_prime = env.get_v_value(current_state)
53     new_q1 = ((1-valeurs.alpha)*q_prime)
54     q_tmp = rew+(valeurs.gamma*v_prime)
55     new_q2 = valeurs.alpha*q_tmp
56     new_q = new_q1 + new_q2
57     env.set_q_value(previous_state, i, new_q)
58     #we set the new V
59     jj = 0
60     bestv = 0
61     bestmv = []
62     while jj < self.nbr_actions() :
63         tmp = env.get_q_value(previous_state, jj)
64         if tmp >= bestv :
65             bestv = tmp
66             bestmv.append(jj)
67         jj += 1
68     #print "bestv = %d"%bestv
69     env.set_v_value(previous_state, bestv)
70     #we set the new pi
71     #jj = 0.
72     #total = 0.
73     #while jj < self.nbr_actions() :
74     #     to_add = env.get_q_value(previous_state, jj)
75     #     total += to_add
76     #     print "Q value %d : %d"%(jj, to_add)
77     #     jj += 1
78     jj = 0
79     l = len(bestmv)
80     while jj < self.nbr_actions() :
81         env.set_pi_value(previous_state, jj, 0)
82         jj += 1
83     for kk in bestmv :
84         env.set_pi_value(previous_state, kk, 1./l)
85     #we set the new alpha

```

```

86     valeurs.alpha = valeurs.alpha*valeurs.decay
87     #decrease exploration coeff
88     valeurs.exploration_coeff = valeurs.exploration_coeff*valeurs.explo
89
90     return depl
91
92 def find_best_move(self,env,agents) :
93     state = env.get_state_name()
94     #ratio d'exploration !
95     expl = valeurs.exploration_coeff
96     val = random.random()
97     #if we are close of the mouse, take it !
98     zem = 0
99     while zem < 5 :
100         for a in agents :
101             if a.test(zem) == 2 :
102                 return zem
103         zem+=1
104
105     if val < expl :
106         mv = random.randrange(self.nbr_actions())
107         return mv
108     else :
109         j = 0
110         r = random.random()
111         step = 0.
112         while j < self.nbr_actions() :
113             #la probabilite de choisir chaque action
114             val = env.get_pi_value(state,j)
115             #print j, val
116             step += val
117             #print j, r, val, step
118             if r <= step :
119                 #print "on a choisi %d" %j
120                 #print "-----"
121                 return j
122             j += 1
123     #soluc = []
124     #maxi = 0
125     #while j < self.nbr_actions() :
126     # val = env.get_pi_value(state,j)
127     # print j, val
128     # if val > maxi :

```

```

129         #         maxi = val
130         #         soluc = [j]
131         #     elif val == maxi :
132         #         soluc.append(j)
133         #     j+=1
134         #r = random.randrange(len(soluc))
135         #print soluc
136         #print "-----"
137         #print "on a choisi %d" %(soluc[r])
138         #print "#####"
139         #return soluc[r]
140
141     def reward(self, target, agents, env):
142         result = target.count(2)
143         murs = target.count(3)
144         reward = 0
145         dist = 0
146         if result == None :
147             print "Erreur dans le return du result"
148         if murs >= 1 :
149             reward -= 1000
150             #print reward
151         if result >= 1 :
152             goal = 1
153             print "GOAL !"
154             reward += 100000
155         else :
156             #reward is the opposite of the distance
157
158             for a in env.get_agents() :
159                 #we want to mesure with agents 2
160                 if a.get_name() == 2 :
161                     #we must initialize the value to a big number
162                     # bigger than the size of the board
163                     tmp = 100000
164                     ennemis = a.get_subagents()
165                     for mouse in ennemis :
166                         tmp2 = 0
167                         for cat in agents :
168                             dis = env.distance(cat.get_pos(), mouse)
169                             if dis == 1 :
170                                 #print "pas loin !"
171                                 #print cat.get_pos(), mouse

```

```

172         reward += 10000
173         #time.sleep(100)
174         # "distance = %d"%dis
175         #time.sleep(5)
176         tmp2 += dis*dis
177         if tmp2 < tmp :
178             tmp = tmp2
179         dist += tmp
180     return 0-dist+reward
181
182 #####
183
184 class obj_souris :
185     def __init__(self , pos) :
186         global valeurs
187         valeurs = values.parameters()
188         self.name = 2
189
190     def nbr_actions(self):
191         return 5
192
193     def action(self , i , agents , env) :
194         if i == 0 :
195             a=agents [0].move(0)
196             return [a]
197         elif i == 1 :
198             a=agents [0].move(1)
199             return [a]
200         elif i == 2 :
201             a=agents [0].move(2)
202             return [a]
203         elif i == 3 :
204             a=agents [0].move(3)
205             return [a]
206         elif i == 4 :
207             a=agents [0].move(4)
208             return [a]
209
210     def find_best_move(self , env , agents):
211         #randomized min-max algorithm for the mouse
212         # here it's the "stochastic best escape" algorithm
213         # we want to stay away of agents 1
214         dic = {}

```

```

215     best = [0,0]
216     moves = []
217     myself = agents[0]
218     j = 0
219     while j < self.nbr_actions() :
220         moves.append(j)
221         j+=1
222     #retirer le pire mouvement
223     for m in moves :
224         #print myself.move(m)
225         # We will delete obviously bad mouvements
226         result = myself.move(m)
227         # here we don't allow move that bring us on the cat
228         if result == 0 :
229             me = agents[0].get_pos()
230             bad = 0
231             for a in env.get_agents() :
232                 #we want to escape agents 1
233                 if a.get_name() == 1 :
234                     tmp = 0.
235                     ennemis = a.get_subagents()
236                     for p in ennemis :
237                         dis = env.distance(me,p)
238                         # we don't accept mvt that go
239                         #to close of a cat
240                         if dis <= 1 :
241                             bad = 1
242                             break
243                         tmp += dis*dis
244                     if bad == 0 :
245                         dic[m] = tmp
246                     if tmp >= best[1] :
247                         best[0] = m
248                         best[1] = tmp
249                 myself.back(m)
250     #print dic
251     somme=0
252     if len(dic) >= 1 :
253         for i in dic :
254             somme+= dic[i]
255     step = 0
256     #now choose a solution randomly
257     # according to the weight of each solution

```

```

258         nbr = random.randrange(somme)
259         for i in dic :
260             step += dic[i]
261             if nbr <= step :
262                 #print i
263                 return i
264     else :
265         #no solution found. We don't move
266         return 0
267
268
269
270 #####
271
272 class obj_env :
273     def __init__(self,xy):
274         global valeurs
275         valeurs = values.parameters()
276         #definition de l'espace
277         #les zeros sont des espaces vides
278         self.space = zeros(xy)

```

ml.py

```
1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3 from livewires import *
4 from Numeric import *
5 from ml_envir import *
6 import random
7
8 class agent :
9     def __init__(self, objet, size, position, env) :
10         self.agents = []
11         self.env = env
12         self.name = objet.name
13         self.size_x = size[0]
14         self.size_y = size[1]
15         for i in position :
16             self.agents.append(single_agent(i, self.name, self.env, self.size))
17         self.me = objet
18
19     def reinit(self) :
20         for i in self.agents :
21             i.reinit()
22
23     def do_action(self, i) :
24         return self.me.action(i, self.agents, self.env)
25
26     def best_move(self) :
27         pos = []
28         b = self.me.find_best_move(self.env, self.agents)
29         return self.do_action(b)
30
31     def get_name(self) :
32         return self.name
33
34     def get_nbr_subagents(self):
35         return len(self.agents)
36
37     def get_subagents(self):
38         pos = []
39         for a in self.agents :
40             pos.append(a.get_pos())
41         return pos
42
```

```

43     def get_nbr_actions(self):
44         return self.me.nbr_actions()
45
46     def get_nbr_states(self):
47         tot = 0
48         for a in self.agents :
49             tot += self.me.nbr_actions()
50         return tot
51
52
53
54
55 class single_agent :
56
57     def __init__(self, position, name, env, size_x, size_y) :
58         self.init = position
59         self.x = position[0]
60         self.y = position[1]
61         self.oldx = position[0]
62         self.oldy = position[1]
63         self.env = env
64         self.name = name
65         self.k = 1
66         self.size_x = size_x
67         self.size_y = size_y
68
69     def reinit(self) :
70         self.x = self.init[0]
71         self.y = self.init[1]
72         self.oldx = self.init[0]
73         self.oldy = self.init[1]
74
75     def get_pos(self) :
76         return [self.x, self.y]
77
78     def test(self, mv) :
79         t = self.move(mv)
80         if t == 0 :
81             self.back(mv)
82         return t
83
84     def move(self, mv) :
85         self.old_x = self.x

```

```

86         self.old_y = self.y
87         if mv == 1 :
88             return self.up()
89         elif mv == 2 :
90             return self.right()
91         elif mv == 3 :
92             return self.bottom()
93         elif mv == 4 :
94             return self.left()
95         else :
96             return self.do_move()
97
98     def back(self, mv) :
99         self.old_x = self.x
100        self.old_y = self.y
101        if mv == 3 :
102            return self.up()
103        elif mv == 4 :
104            return self.right()
105        elif mv == 1 :
106            return self.bottom()
107        elif mv == 2 :
108            return self.left()
109        else :
110            return self.do_move()
111
112     def left(self) :
113         self.x = self.x - self.k
114         return self.do_move()
115
116     def right(self) :
117         self.x = self.x + self.k
118         return self.do_move()
119
120     def up(self) :
121         self.y = self.y + self.k
122         return self.do_move()
123
124     def bottom(self) :
125         self.y = self.y - self.k
126         return self.do_move()
127
128     def do_move(self):

```

```

129     if self.x >= self.size_x :
130         self.x = self.x - self.size_x
131     if self.y >= self.size_y :
132         self.y = self.y - self.size_y
133     if self.x < 0 :
134         self.x = self.x + self.size_x
135     if self.y < 0 :
136         self.y = self.y + self.size_y
137
138     target = self.env.get([self.x, self.y])
139     #print "target = %s"%str(target)
140     if target == 0 :
141         self.env.remove([self.old_x, self.old_y])
142         self.env.put(self.name, [self.x, self.y])
143         return 0
144     elif self.x == self.old_x and self.y == self.old_y :
145         return 0
146     else :
147         self.x = self.old_x
148         self.y = self.old_y
149         return target
150     # visualisation
151     #move_to(self.me, self.x, self.y)

```

ml_envir.py

```
1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3 from livewires import *
4 from Numeric import *
5 import ml
6 import random
7
8
9 class envir :
10     def __init__(self, envi) :
11         self.space = envi.space
12         self.agents=[]
13         self.nbr_actions = 0
14         self.nbr_states = 0
15         self.init ={}
16         # Q is a dictionnary of dictionnaries
17         # keys are the state name
18         # entry is a dictionnary for available actions
19         self.Q = {}
20         self.V = {}
21         self.pi= {}
22         self.size = [len(self.space),len(self.space[0])]
23         #for a in agents :
24             #ici on a le nombre d'actions possible
25             # self.nbr_actions += a.get_nbr_actions()
26             # self.nbr_states += a.get_nbr_states()
27
28     def get_agents(self):
29         return self.agents
30
31     def get(self,xy):
32         return self.space[xy[0]][xy[1]]
33
34     def reinit(self):
35         for a in self.agents :
36             a.reinit()
37         self.space = zeros(self.size)
38         for i in self.init :
39             for j in self.init[i] :
40                 self.put(i, j)
41
42     def put(self, objet, coord) :
```

```

43     x = coord[0]
44     y = coord[1]
45     value = self.space[x][y]
46     self.space[x][y] = objet
47     return value
48
49 def remove(self , coord) :
50     x = coord[0]
51     y = coord[1]
52     value = self.space[x][y]
53     self.space[x][y] = 0
54     return value
55
56 def addagent(self , objet , positions):
57     name = objet.name
58     self.init[name] = positions
59     for a in positions :
60         self.put(name,a)
61     new_ag = ml.agent(objet , self.size , positions , self)
62     self.agents.append(new_ag)
63     # est-ce que l'agent compte dans les actions possibles ?
64     if name == 1 :
65         self.nbr_actions += new_ag.get_nbr_actions()
66         self.nbr_states += new_ag.get_nbr_states()
67     return new_ag
68
69
70 def get_state_name(self):
71     name = ''
72     for i in self.space :
73         for j in i :
74             name+=str(j)
75     #now we can take advantage of the symetry
76     #we will remove all leading and trailing 0
77     #this is optional to improve performance A LOT
78     #we must always start with the mouse (2 in our example)
79     b=name.split('2',1)
80     final='2'
81     final+=b[1]
82     final+=b[0]
83     name = final.strip('0')
84     #then
85     return name

```

```

86
87 def distance(self ,a,b) :
88     k_x = self.size[0]
89     k_y = self.size[1]
90     #print "distance :"
91     #print a,b
92     distance = 0
93     ### X###
94     if a[0] < b[0] :
95         horiz1 = b[0] - a[0]
96         horiz2 = k_x - b[0] + a[0]
97         if horiz1 < horiz2 :
98             distance += horiz1
99         else :
100             distance += horiz2
101     elif a[0] > b[0] :
102         horiz1 = a[0] - b[0]
103         horiz2 = k_x - a[0] + b[0]
104         if horiz1 < horiz2 :
105             distance += horiz1
106         else :
107             distance += horiz2
108     #print "en X"
109     #print distance
110     ### y ###
111     if a[1] < b[1] :
112         horiz1 = b[1] - a[1]
113         horiz2 = k_y - b[1] + a[1]
114         if horiz1 < horiz2 :
115             distance += horiz1
116         else :
117             distance += horiz2
118     elif a[1] > b[1] :
119         horiz1 = a[1] - b[1]
120         horiz2 = k_y - a[1] + b[1]
121         if horiz1 < horiz2 :
122             distance += horiz1
123         else :
124             distance += horiz2
125     #print "total"
126     #print distance
127     return distance
128

```

```

129     def get_q_value(self , state , action):
130     # here , Q is virtually filled with 1
131         try :
132             value = self.Q[state][action]
133             return value
134         except :
135             return 1
136
137     def set_q_value(self , state , action , value):
138         etat = str(state)
139         try :
140             self.Q[etat]
141         except :
142             dic={}
143             self.Q[etat] = dic
144             self.Q[etat][action] = value
145
146
147     def get_v_value(self , state):
148     # here , V is virtually filled with 1
149         try :
150             value = self.V[state]
151             return value
152         except :
153             return 1
154
155     def set_v_value(self , state , value) :
156         self.V[state] = value
157
158     def get_pi_value(self , state , action):
159     # here , pi is virtually filled with 1/A
160         try :
161             value = self.pi[state][action]
162             return value
163         except :
164             if self.nbr_actions == 0 :
165                 return 0
166             else :
167                 #nbr of
168                 return 1./self.nbr_actions
169
170     def set_pi_value(self , state , action , value):
171         etat = str(state)

```

```

172         try :
173             self.pi[etat]
174         except :
175             dic={}
176             self.pi[etat] = dic
177             self.pi[etat][action] = value
178
179     def display(self) :
180         return self.space
181
182     def q_infos(self) :
183         total = 0
184         maxi = 0
185         for i in self.Q :
186             l = len(self.Q[i])
187             total += l
188             if l > maxi :
189                 maxi = l
190         print "longueur de Q : %d ( %d actions testees max par etat)"%(total,maxi)
191         return maxi
192         #print self.Q
193
194     def v_infos(self) :
195         print "longueur de V : %d"%len(self.V)
196         #print self.init
197         #print self.V
198
199     def pi_infos(self) :
200         total = 0
201         for i in self.pi :
202             total += len(self.pi[i])
203         print "longueur de pi : %d"%total
204         #print self.pi

```

display.py

```
1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3 from livewires import *
4 import values
5
6 #this is a display implementation. Simply implement a show(matrice) method
7 # the env matrice as an argument.
8 def show(matrice) :
9     valeurs = values.parameters()
10    clear_screen()
11    i = 0
12    while i <= valeurs.size_x :
13        j=0
14        while j <= valeurs.size_y :
15            box(i,j,i+valeurs.k,j+valeurs.k)
16            j += valeurs.k
17        i += valeurs.k
18    allow_moveables()
19    def place(x,y,color):
20        size = 0.4*valeurs.k
21        posx = (x+0.5)*valeurs.k
22        posy = (y+0.5)*valeurs.k
23        #if color == 0 :
24        # posx = posx - (valeurs.k*0.5)
25        # posy = posy - (valeurs.k*0.5)
26        # o=box(x,y,x+valeurs.k,y+valeurs.k, filled=0)
27        #move_to(o, posx, posy)
28        if color == 1 :
29            o=circle(posx,posy,size,filled=2)
30            #move_to(o, posx, posy)
31        if color == 2 :
32            o=circle(posx,posy,size,filled=0)
33            #move_to(o, posx, posy)
34        if color == 3 :
35            posx = posx - (valeurs.k*0.5)
36            posy = posy - (valeurs.k*0.5)
37            o=box(posx,posy,posx+valeurs.k,posy+valeurs.k, filled=1)
38            #move_to(o, posx, posy)
39    ii = 0
40    while ii < len(matrice) :
41        jj = 0
42        while jj < len(matrice[ii]) :
```

```
43         place(ii ,jj ,matrice [ ii ][ jj ])
44         jj += 1
45     ii += 1
46 o = circle (0,0,0, filled=0)
47 move_to(o,0,0)
48 time.sleep(valeurs.to_sleep)
```

jeu1.py

```
1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3 from livewires import *
4 import random
5
6 class agent :
7     def __init__(self ,x,y,color) :
8         self.size_x = size_x
9         self.size_y = size_y
10        self.k = k
11        size = k*0.4
12        posx = (x+0.5)*k
13        posy = (y+0.5)*k
14        print posx ,posy
15        self.me = circle (posx ,posy , size , filled=color)
16        self.x = posx
17        self.y = posy
18        self.do_move()
19
20    def coord(self) :
21        return [self.x ,self.y]
22
23    def move(self , mv) :
24        if mv == 1 :
25            self.up()
26        elif mv == 2 :
27            self.right()
28        elif mv == 3 :
29            self.bottom()
30        elif mv == 4 :
31            self.left()
32        else :
33            self.do_move()
34
35    def back(self , mv) :
36        if mv == 3 :
37            self.up()
38        elif mv == 4 :
39            self.right()
40        elif mv == 1 :
41            self.bottom()
42        elif mv == 2 :
```

```

43         self.left()
44     else :
45         self.do_move()
46
47     def left(self) :
48         self.x = self.x - self.k
49         self.do_move()
50
51     def right(self) :
52         self.x = self.x + self.k
53         self.do_move()
54
55     def up(self) :
56         self.y = self.y + self.k
57         self.do_move()
58
59     def bottom(self) :
60         self.y = self.y - self.k
61         self.do_move()
62
63     def do_move(self):
64         if self.x >= self.size_x :
65             self.x = self.x - self.size_x
66         if self.y >= self.size_y :
67             self.y = self.y - self.size_y
68         if self.x < 0 :
69             self.x = self.x + self.size_x
70         if self.y < 0 :
71             self.y = self.y + self.size_y
72         move_to(self.me, self.x, self.y)
73
74     class environnement :
75         def __init__(self, agents, objectif) :
76             self.agents=agents
77             self.obj = objectif
78             self.size_x = size_x
79             self.size_y = size_y
80             self.k = k
81
82     def goal(self):
83         goal = 0
84         for i in self.agents :
85             if i.coord()[0] == self.obj.coord()[0] and i.coord()[1] == self

```

```

86             goal = 1
87         return goal
88
89     def value_agent(self , agent) :
90         tmp = self.distance(agent.coord() , self.obj.coord())
91         return 0-tmp
92
93     def reward(self , agent):
94         if self.goal() == 1 :
95             return 10000
96         else :
97             return self.value_agent(agent)
98
99     def exp_return(self , a, depth) :
100         max_depth = 4
101         if depth > max_depth :
102             return self.reward(a)
103         d = depth+1
104         gamma = 0.9
105         move = 0
106         #move other agents
107         #move the mouse according to probable move
108         value = self.exp_return(a,d)
109         a.right()
110         right = self.exp_return(a,d)
111         a.left()
112         if right > value :
113             move = 2
114             value = right
115         a.left()
116         left = self.exp_return(a,d)
117         a.right()
118         if left > value :
119             move = 4
120             value = left
121         a.up()
122         up = self.exp_return(a,d)
123         a.bottom()
124         if up > value :
125             move = 1
126             value = up
127         a.bottom()
128         bot = self.exp_return(a,d)

```

```

129         a.up()
130         if bot > value :
131             move = 3
132             value = bot
133
134         return self.reward(a)+gamma*value
135
136     #randomized min-max algorithm for the mouse
137     def best_escape(self) :
138         depth=2
139         m = self.obj
140         array=[0,0,0,0,0]
141         def calculpos(mv) :
142             m.move(mv)
143             rew = 0
144             for a in self.agents :
145                 tmp = self.exp_return(a,depth)
146                 rew += (tmp*tmp)
147             m.back(mv)
148             return rew
149         for i in [0,1,2,3,4] :
150             array[i] = calculpos(i)
151         mini = max(array)
152         index = array.index(mini)
153         print array , index
154         return index
155
156
157     def best_move(self , agent) :
158         best = 0
159         same = 0
160         actual = self.exp_return(agent,1)
161         agent.up()
162         up = self.exp_return(agent,1)
163         agent.bottom()
164         if up > actual :
165             best = 1
166             actual = up
167         elif up != actual :
168             same = 1
169         agent.bottom()
170         bot = self.exp_return(agent,1)
171         agent.up()

```

```

172         if bot > actual :
173             best = 3
174             actual = bot
175         elif bot != actual :
176             same = 1
177         agent.left()
178         left = self.exp_return(agent,1)
179         agent.right()
180         if left > actual :
181             best = 4
182             actual = left
183         elif left != actual :
184             same = 1
185         agent.right()
186         right = self.exp_return(agent,1)
187         agent.left()
188         if right > actual :
189             best = 2
190             actual = right
191         elif right != actual :
192             same = 1
193         #if same == 0 :
194         # best = random.randint(0,4)
195         print "up: %d -bot: %d -right: %d -left: %d = %d"%(up,bot ,right ,left,best)
196         #print "agent : "+agent.coord()
197         #print "obj :"+self.obj.coord()
198         return best
199
200
201     def distance(self ,a,b) :
202         k_x = self.size_x
203         k_y = self.size_y
204         distance = 0
205         ### X###
206         if a[0] < b[0] :
207             horiz1 = b[0] - a[0]
208             horiz2 = k_x - b[0] + a[0]
209             if horiz1 < horiz2 :
210                 distance += horiz1
211             else :
212                 distance += horiz2
213         elif a[0] > b[0] :
214             horiz1 = a[0] - b[0]

```

```

215         horiz2 = k_x - a[0] + b[0]
216         if horiz1 < horiz2 :
217             distance += horiz1
218         else :
219             distance += horiz2
220     ### y ####
221     if a[1] < b[1] :
222         horiz1 = b[1] - a[1]
223         horiz2 = k_y - b[1] + a[1]
224         if horiz1 < horiz2 :
225             distance += horiz1
226         else :
227             distance += horiz2
228     elif a[1] > b[1] :
229         horiz1 = a[1] - b[1]
230         horiz2 = k_y - a[1] + b[1]
231         if horiz1 < horiz2 :
232             distance += horiz1
233         else :
234             distance += horiz2
235     return distance
236
237
238 if __name__ == "__main__":
239     global size_x , size_y , k
240     size_x = 200
241     size_y = 200
242     k = 20
243     begin_graphics(width=size_x , height=size_y)
244     i = 0
245     while i <= size_x :
246         j=0
247         while j <= size_y :
248             box(i , j , i+k , j+k)
249             j += k
250         i += k
251     allow_moveables()
252     #time.sleep(3)
253     cat1 = agent(1,1,1)
254     cat2 = agent(7,3,1)
255     cat3 = agent(4,4,4)
256     time.sleep(3)
257     mouse = agent(6,7,0)

```

```

258     time.sleep(3)
259     env = environnement([cat1, cat2], mouse)
260     i = 10
261     moves = 0
262
263     while env.goal() == 0 :
264         print "Mouse try to escape"
265         mouse.move(env.best_escape())
266         time.sleep(0.5)
267         print "cat try to catch the mouse"
268         cat1.move(env.best_move(cat1))
269         cat2.move(env.best_move(cat2))
270         cat3.move(env.best_move(cat3))
271         moves += 1
272         #print "#####"
273         #print i
274         #print "#####"
275         #cat1.right()
276         #time.sleep(0.5)
277         #mouse.left()
278         #time.sleep(0.5)
279         #cat1.bottom()
280         #time.sleep(0.5)
281         #mouse.up()
282         #i += 1
283         #print "#####"
284         #print env.value_agent(cat1)
285         time.sleep(0.5)
286     end_graphics()
287     print "%d mouvements pour attraper la souris" %moves

```