ÉCOLE SUPÉRIEURE D'INGÉNIEURS
EN ÉLECTRONIQUE
ET ÉLECTROTECHNIQUE

RLtoolbox V.1 : Boîte à Outils sous Scilab d'Apprentissage Par Renforcement

T. AL-ANI

Laboratoire A2SI

22 Octobre 2003

Contexte de la boîte à outil RLtoolbox

Cette boîte à outils (*RLtoolbox*)implémente différents RL algorithmes sous Scilab[1].

Nous pouvons définir l'apprentissage par renforcement (Reinforcement Learning (RL))[2][3] comme suit :

- Apprentissage par interaction
- Apprentissage type orienté-objectif (Goal-oriented learning)
- Apprentissage basé sur l'interaction avec l'environnement externe (apprendre sur, de, et pendant l'interaction avec un environnement externe)
- Apprendre quoi faire, comment associer des situations à des actions pour qu'une valeur de récompense numérique soit maximisé.

Cette boîte à outils est surtout destinée à un objectif de simulation et pourra être utilisée comme un outil pédagogique. En outre, elle devra faire preuve d'un maximum de convivialité. Plusieurs exemples de démonstration sont implémentés (des jeux, robot mobile et autres processus physiques simulés). Plusieurs approches sont intégrées dans cette boîte :

1. Programmation dynamique
2. Monte Carlo
3. Apprentissage par la méthode de Différence Temporelle *"Temporal Difference Learning"*

[1] Un logiciel de calcul scientifique : `http ://www-rocq.inria.fr/scilab/`

[2] `http ://www-anw.cs.umass.edu/~rich/book/the-book.html`

[3] `http ://www.esiee.fr/~info/a2si/docu_ens.html`

Table des matières

Chapitre 1

Introduction à l'apprentissage par renforcement

1.1 Brève comparaison entre l'apprentissage supervisé et l'apprentissage par renforcement

Dans le cas de l'apprentissage supervisé, l'entrainement se fait en donnant à l'entité devant apprendre[1] une entrée et le résultat qui devrait être théoriquement obtenu. L'agent cherchera donc à minimiser l'erreur représentée par la différence entre la sortie trouvée et la sortie qui aurait dû être trouvée.

Dans le cas de l'apprentissage par renforcement, l'entité apprenante est autonome et s'entraîne à partir des entrées et des récompenses (ou pénalités) associées pour décider de l'action à effectuer. L'agent cherchera donc à maximiser les récompenses.

Une des premières différence importante est donc le fait que l'agent apprenant par renforcement interagit avec son environnement : il reçoit des informations de celui-ci et peut le modifier. Ces spécificités seront développpées par la suite. La seconde grande différence est l'autonomie. En effet, un agent utilisant l'apprentissage supervisé doit connnaitre les résultats théoriques pour parfaire son entraînement.

Maintenant que nous avons vu les deux principales méthodes d'apprentissages qui ont été informatisées nous allons étudier plus en détails ce qu'est l'apprentissage par renforcement.

1.2 Description de l'apprentissage par renforcement

1.2.1 Généralités

Le problème posé

L'apprentissage par renforcement est une approche programmable. Elle permet à un agent d'atteindre un but à long terme sans qu'on lui ait spécifié comment remplir sa tâche. Pour cela, il interagit avec son environement et associe des **bonus/malus** aux différentes situations obtenues. L'agent cherchera donc à atteindre son but en se basant sur la maximisation des récompenses.

Il doit pour ce faire passer par des phases d'**exploration** et des phases d'**exploitation**, le problème étant de trouver le bon équilibre entre chacunes. En effet, pour augmenter la récom-

[1] que l'on appellera agent

pense, l'agent doit utiliser des actions qu'il a déjà essayé (donc dont il connait la récompense) : **c'est l'exploitation**. Cette phase permet aussi d'ajuster la valeur des actions parcourues (dont les récompenses peuvent variées d'où la nécessité de les tester plusieurs fois). Cependant il faut aussi **explorer** les autres possibilités afin d'ajuster aussi leur valeur, ce qui peut amener l'agent à trouver une meilleure suite d'actions permettant d'atteindre son objectif. On voit donc la nécessité de trouver un bon équilibre entre les phases d'**exploration** et les phases d'**exploitation**.

Maintenant que nous avons effectué quelques généralités, nous allons voir les éléments caractéristiques qui peuvent être extraits.

Eléments caractéristiques

Avec l'agent et l'environnement, on peut identifier plusieurs éléments intervenants dans un problème d'apprentissage par renforcement :

- **la politique :** elle détermine le comportement de l'agent à un instant donné, c'est une association entre un état donné de l'environnement et l'action à effectuer dans cet état.
- **la récompense/pénalité :** indique la désirabilité d'un état.
- **la fonction valeur :** spécifie ce qui est bien dans l'avenir. Cette fonction indique la désirabilité d'un état en tenant compte de la valeur des états suivants celui-ci.
- **le modèle de l'environnement :** mime le comportement de l'environnement.

1.2.2 Quelques précisions

Définitions et Notations

L'agent interagit avec l'environnement à des pas de temps discrets. A chaque pas de temps t, il reçoit une représentation de l'état de l'environnement $s_t \in S$, où S est l'ensemble des états possibles, et à partir de cela choisit une action $a_t \in A(s_t)$, où $A(s_t)$ est l'ensemble des actions disponibles sous l'état s_t. Au pas suivant, en partie à cause de son action, l'agent reçoit une récompense numérique, $r_{t+1} \in \Re$, et se retrouve dans un nouvel état $s+1$:

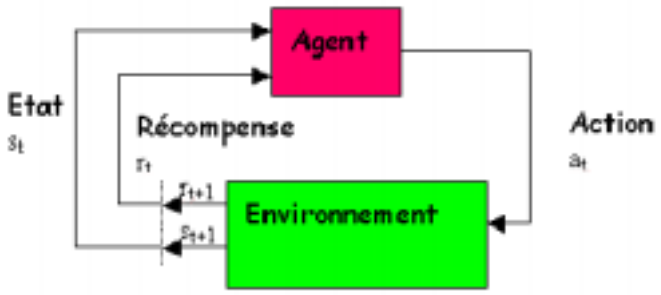


FIG. 1.1 – Interaction Agent-Environnement

Par la suite, on définit :

- *la politique au pas t,* π *:* $\pi_t(s,a) = Pr(a_t = a/s_t = s)$ avec $a = action$ et s =état
- *le revenu* R_t *:* fonction d'une séquence de récompenses et du type de tâches effectuées.
- *la valeur d'un état* $V(s)$ *:* revenu espéré en partant de cet état, liée à la politique de l'agent
- *la valeur du choix d'une action* a *dans un état* s *sous une politique* π*,* $Q_\pi(s,a)$ *:* revenu espéré en partant de l'état s, en choisissant l'action a, et en poursuivant la politique π

Les tâches et leur revenu

On peut distinguer deux types de tâche :
- *tâches épisodiques :* l'interaction avec l'environnnement se sépare naturellement en épisodes (parties d'un jeu, parcours d'un labyrinthe, ...)
- *tâches continues :* l'interaction n'a pas d'épisode naturel (robot explorant une planète, ...)

Cela nous amène donc à deux fonctions de récompenses différentes :
- *pour les tâches épisodiques :*

$$R_t = r_{t+1} + r_{t+2} + ... + r_T \tag{1.1}$$

avec T le pas de temps final auquel on atteint un état terminal, la fin d'un épisode
- *pour les tâches continues :*

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{1.2}$$

avec $\gamma, 0 \leq \gamma \leq 1$, est le taux d'escompte. Suivant la valeur de γ l'agent adopte une vision à plus ou moins long terme : *court terme* $0 \leftarrow \gamma \rightarrow 1$ *long terme*

Cependant, il est possible d'utilser la formule générale suivante :

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{1.3}$$

avec γ qui est égal à 1 si un état absorbant avec une récompense nulle est toujours atteinte (cas des tâches épisodiques).

Propriété et Processus de Décision de Markov

Une tâche d'apprentissage par renforcement a la propriété de Markov si un état rassemble les observations ou les sensations antérieures pour ne retenir que l'information essentielle. Cette tâche est appelée Processus de Décision de Markov. Les probabilités de transition et de récompense sont donc les suivantes :

$$P^a_{ss'} = Pr\{s_{t+1} = s' | s_t = s, a_t = a\}, R^a_{ss'} = E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s\} \tag{1.4}$$

avec s, s' appartenant à S ensemble des états et a appartenant à $A(s)$ l'ensemble des actions.

Valeur d'un état et valeur du choix d'une action

La valeur d'un état est le retour attendu en partant de cet état. Il dépend donc de la politique de l'agent :

Fonction Valeur d'Etat pour la Politique π

$$V^\pi(s) = E_\pi\{R_t | s_t = s\} = E_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\} \tag{1.5}$$

$$V^\pi(s) = \sum_a \pi(s,a) \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^\pi(s')] \tag{1.6}$$

Pour les Processus de Décisions de Markov (MDP en Anglais) fini (c'est à dire avec un nombre d'états et d'actions fini), les politiques peuvent être partiellement ordonnées :

$$\pi \geq \pi' \Leftrightarrow V^{\pi}(s) \geq V^{\pi'}(s) \ \forall s \in S \tag{1.7}$$

Il y a toujours au moins une (et il peut y en avoir plusieurs) politique qui est meilleure ou égale à toutes les autres. C'est (ou ce sont) la (ou les) politique(s) optimale(s) que l'on notera π^*. Dans le cas où il y aurait plusieurs politiques optimales, il est important de noter qu'elles partagent la même fonction valeur d'état :

$$V^*(s) = \max_{\pi} V^{\pi}(s) \tag{1.8}$$

La valeur correspondant au choix d'une action dans un état avec une politique π est le retour attendu en partant de cet état en prenant cette action et en suivant la politique π :

Fonction Valeur d'Action pour la Politique π

$$Q^{\pi} = E_{\pi}\{R_t | s_t = s, a_t = a\} = E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+l+1} | s_t = s, a_t = a\} \tag{1.9}$$

Tout comme pour la fonction valeur d'état, dans le cas où il y aurait plusieurs politiques optimales, elles partagent la même fonction valeur d'action :

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) \ \forall s \in S \ et \ \forall a \in A(s) \tag{1.10}$$

$V^* et Q^*$ sont les **solutions uniques du système d'équations non linéaires suivantes :**

$$V^*(s) = \max_{a \in A(s)} \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^*(s')] \tag{1.11}$$

$$Q^*(s,a) = \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma \max_{a'} Q^*(s',a')] \tag{1.12}$$

Comme conséquences directes nous avons donc que :

toute politique gloutonne qui respecte V^* est une politique optimale

$$\pi^*(s) = arg \max_{a \in A(s)} Q^*(s,a) \tag{1.13}$$

1.3 Rétroaction évaluative

1.3.1 Le problème du n-armed bandit

Dans ce type de problème, à chaque tour il y a n choix possibles, et chaque choix est appelé une action. Pour chaque action on a une récompense, qui au départ est inconnue. L'objectif est de maximiser la récompense sur le long terme (plus de 1 000 jeux). Pour résoudre un tel problème l'apprentissage par renforcement est très interessant, en effet il faut à la fois explorer de nouvelles actions (pour en connaitre les récompenses associées), mais également exploiter les connaissances des jeux précédents.

Méthode par action-valeur On utilise ici comme approximation d'une valeur d'une action après t expériences une moyenne des récompenses obtenues pour cette action, définies par la formule :

$$Q_t(a) = \frac{r_1 + r_2 + ... + r_{k_a}}{k_a} \tag{1.14}$$

On pourra noter que

$$\lim_{k_a \to \infty} Q_t(a) = Q^*(a) \tag{1.15}$$

où $Q^*(a)$ est la valeur théorique.

Sélection gloutonne On note a_t l'action qui sera sélectionnée, en l'occurence celle qui aura la plus valeur Q_t.La sélection gloutonne se base donc sur la formule :

$$a_t = a_t^* = arg\max_a Q_t(a) \tag{1.16}$$

On peut ensuite introduire la méthode ϵ-gloutonne qui va choisir à chaque étape entre la méthode gloutonne (avec une probabilité $1 - \epsilon$) et un choix d'exploration aléatoire (avec une probabilité ϵ). Le choix entre exploration et exploitation est donc complètement basé sur la valeur d'ϵ.

Softmax sélection Cette méthode de sélection se base sur des probabilités pour décider quelles actions explorer en priorités. Le choix de l'action a au jeu t a la probabilité :

$$\frac{e^{Q_t(a)/\tau}}{\sum_{b=1}^{n} e^{Q_t(b)/\tau}} \tag{1.17}$$

La variable τ permet de sélectionner la 'température' des choix : plus τ est proche de 0, plus on tend vers la méthode gloutonne classique, et plus τ tend vers 1 plus on accentue les différences entre les actions.

1.3.2 Le problème du bandit binaire

On s'intéresse maintenant au problème ou seulement deux actions sont possibles (1 ou 2) et uniquement deux récompenses (succès ou échec).

Apprentissage linéaire par automates Il existe deux méthodes d'apprentissage linéaire, l'une basée uniquement sur la récompense (on l'appelera L_{R-I}) et l'autre sur la récompense et la pénalité (on l'appelera L_{R-P}). Ces méthodes mettent à jour leurs probabilités à chaque action. Si l'action sélectionnée est un succès, la probabilité de choisir l'action est incrémentée selon la formule :

$$\pi_{t+1}(a_t) = \pi_t(a_t) + \alpha(1 - \pi_t(a_t)) \tag{1.18}$$

avec

$$0 < \alpha < 1$$

Les autres probalités sont ajustées pour que la somme des probabilités soit toujours égales à un. En cas d'échec, on ne fait rien dans la méthode L_{R-I} et pour la méthode L_{R-P} on met à jour les probabilités par la formule :

$$\pi_{t+1}(a_t) = \pi_t(a_t) + \alpha(0 - \pi_t(a_t)) \tag{1.19}$$

avec

$$0 < \alpha < 1$$

Retour sur la méthode action-valeur Pour calculer la nouvelle valeur d'un état selon cette méthode on doit faire la moyenne des récompenses, ce qui impliquerait de stocker toutes les récompenses obtenues. Afin d'éviter de conserver toutes ces récompenses, on pourra utiliser la formule suivante :

$$Q_{k+1}(a) = Q_k(a) + \frac{1}{k+1}[r_{k+1} - Q_k] \tag{1.20}$$

On reconnait alors la formule générale de mise à jour :

$$NouvelleEstimation = AncienneEstimation + Pas * [But - AncienneEstimation] \tag{1.21}$$

On pourra noter que $[But - AncienneEstimation]$ représente 'l'erreur' par rapport à l'estimation.

Problème non stationnaire Toutes les méthodes vues précédemments sont appropriées pour un problème stationnaire, c'est à dire qui n'est pas modifié au cours du temps. Cependant de nombreux problèmes sont non stationnaires. En particulier les méthodes par moyenne ne peuvent pas convenir sur un problème qui se modifie au cours du temps. On va donc accorder plus de poids aux récompenses récentes, on a alors une nouvelle formule pour la probabilité de choisir un état :

$$Q_k = (1-\alpha)^k Q_0 + \sum_{i=1}^{k} \alpha(1-\alpha)^{k-i} r_i \tag{1.22}$$

Choix des valeurs initiales Ce choix est primordial pour l'évolution de l'algorithme. Il sera souvent plus interessant de mettre des valeurs initiales optimistes qui vont permettre d'atteindre plus rapidement les actions optimales(malgré un départ plus difficile). Mais ceci dépend fortement de l'environnement.

Comparaison des renforcement Le système d'apprentissage par renforcement se base sur les valeurs des récompenses, il se pose alors comme problème de décider si une récompense sera considérée comme bonne ou mauvaise. L'idée est de comparer la récompense à la moyenne des récompenses obtenues.
On a alors pour la probabilité d'une action par Gibbs :

$$\frac{e^{P_t(a)}}{\sum_{b=1}^{n} e^{P_t(b)}} \tag{1.23}$$

$$p_{t+1}(a_t) = p_t(a) + [r_t - \bar{r}_t] \tag{1.24}$$

$$\bar{r}_{t+1} = \bar{r}_t + \alpha[r_t - \bar{r}_t] \tag{1.25}$$

Méthode de poursuite Cette méthode utilise à la fois l'estimation par action-valeur et la préférence des actions, le but étant de rendre le choix glouton de plus en plus choisi au cours du temps. L'action gloutonne est alors définie par : $a^*_{t+1} = argmax_a Q_{t+1}(a)$

On alors la probabilité de choisir l'état a^* :

$$\pi_{t+1}(a^*_{t+1}) + \beta[1 - \pi_t(a^*_{t+1})] \tag{1.26}$$

Les autres probabilités sont décrémentées par la formule :

$$\pi_{t+1}(a_{t+1}) + \beta[0 - \pi_t(a^*_{t+1})] \tag{1.27}$$

Chapitre 2

Programmation Dynamique

2.1 La théorie

2.1.1 Evaluation de politique

L'évaluation d'une politique π consiste à calculer sa fonction valeur d'état V^π. Ce calcul peut s'effectuer grâce au fait que :

$$\forall s \in S, V^\pi(s) = \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^\pi(s')\right] \tag{2.1}$$

Si la dynamique de l'environnement est entièrement connu, alors l'équation précédente se résume à un système linéaire de $|S|$ équations avec autant d'inconnus. Il est possible de résoudre ce système de manière classique, mais il est plus judicieux d'utiliser une méthode itérative où l'on choisit un V_0 arbitrairement, et on fait évoluer la solution par :

$$V_{k+1}(s) = \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V_k(s')\right] \tag{2.2}$$

et ce, pour tout $s \in S$. On peut montrer qu'en général, la suite $\{V_k\}$ converge vers V^π. Par ailleurs, pour passer de l'itération k à l'itération $k+1$, on applique les mêmes opérations à chaque état s : on remplace l'ancienne valeur de s par une nouvelle obtenue à partir des états successeurs de s et de la récompense immédiate due à cette transition d'état suivant la politique π. On appelle ce type d'opération un *full backup*.

Algo. 2.1 Algorithme d'évaluation itérative de politique

1. Input π, the policy to be evaluated
   Initialize $V(s) = 0$, for all $s \in S^+$

2. Repeat
   $\Delta \leftarrow 0$
   For each $s \in S$
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V(s')\right]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   Until $\Delta < \theta$ (a small positive number)

3. Output $V \approx V^\pi$

Du point de vue de l'implémentation de cet algorithme, il faudra donc utiliser deux tableaux, un pour les anciennes valeurs $V_k(s)$ et un pour les nouvelles valeurs $V_{k+1}(s)$. De plus, il faut bien que l'algorithme s'arrête avant d'atteindre l'infini, donc on va calculer l'écart entre deux itérations successives, $\max_{s\in S} |V_{k+1}(s) - V_k(s)|$, et on s'arrête quand cet écart est suffisamment petit. L'algorithme complet est donné en 2.1 (page 13).

2.1.2 Amélioration de politique

Un autre problème fort intéressant est de déterminer à partir d'une fonction valeur V^π connue d'une politique déterministe arbitraire π, si il est possible pour un état s de choisir une action $a \neq \pi(s)$ et ainsi améliorer la politique. $V^\pi(s)$ nous renseigne sur la désirabilité d'un état, mais en suivant tout le temps la politique π. En revanche, $Q^\pi(s,a)$ nous donne la désirabilité d'un état en effectuant l'action a à partir de cet état, puis en suivant la politique π pour les prochaines décisions. Nous rappelons que :

$$Q^\pi(s,a) = \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^\pi(s')\right] \tag{2.3}$$

Appellons π' la politique identique à la politique π sauf pour l'état s où on choisit a au lieu de $\pi(s)$. On a alors que la politique π' est meilleure (au sens large) que la politique π si on a :

$$Q^\pi(s,\pi'(s)) \geq V^\pi(s) \tag{2.4}$$

Ceci est la conséquence directe du théorème d'amélioration de politique qui dit que pour tout couple de politique déterministes π et π', π' est aussi bonne voire meilleure que π (i.e. $\pi' \geq \pi$) si pour tout $s \in S$,

$$Q^\pi(s,\pi'(s)) \geq V^\pi(s) \tag{2.5}$$

Enfin, si la politique π' est aussi bonne mais pas meilleure que π, alors $V^\pi = V^{\pi'}$, et comme on a une politique gloutonne pour π', on a pour tout $s \in S$:

$$V^{\pi'}(s) = \max_a \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^{\pi'}(s')\right] \tag{2.6}$$

2.1.3 Itération de la politique

Etant donné ce que l'on vient de voir, une conséquence simple et directe et de partir d'une politique π, de l'évaluer en calculant V^π qui sert à trouver une meilleure politique π' pour laquelle on peut aussi calculer $V^{\pi'}$. On peut donc utiliser la séquence suivante pour trouver une politique optimale $\pi*$:

$$\pi_0 \xrightarrow{E} V^{\pi_0} \xrightarrow{i} \pi_1 \xrightarrow{E} V^{\pi_1} \xrightarrow{i} \pi_2 \xrightarrow{E} \ldots \xrightarrow{I} \pi^* \xrightarrow{E} V^* \tag{2.7}$$

L'algorithme 2.2 (page 15) permet donc de trouver une politique optimale en un nombre d'itérations généralement très faible.

2.1.4 Itération de valeur

L'itération de la politique présente l'inconvénient de demander l'évaluation de la politique à chaque itération. Cette évaluation est aussi un calcul itératif. Cependant, en utilisant la propriété suivante :

$$V_{k+1}(s) = \max_a \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V_k(s')\right] \tag{2.8}$$

on peut obtenir un algorithme donné en 2.3 (page 15) qui permet d'obtenir une solution équivalente.

ALGO. 2.2 Algorithme de politique itérative

1. Initialization
   $V(s) \in \Re$ and $\pi(s) \in A(s)$ arbitrarly for all $s \in S$

2. Policy evaluation
   Repeat
   $\Delta \leftarrow 0$
   For each $s \in S$
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s'} P_{ss'}^{\pi(s)} \left[R_{ss'}^{\pi(s)} + \gamma V(s') \right]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   Until $\Delta < \theta$ (a small positive number)

3. Policy improvement
   $policy - stable \leftarrow true$
   for each $s \in S$
   $b \leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s'} P_{ss'}^a \left[R_{ss'}^a + \gamma V(s') \right]$
   if $b \neq \pi(s)$ then $policy - stable \leftarrow false$
   If $policy - stable$, then stop ; else go to 2

ALGO. 2.3 Algorithme d'itération valeurs

1. Initialize v arbitrarily, e.g., $V(s) = 0$ for all $s \in S^+$

2. Repeat
   $\Delta \leftarrow V(s)$
   For each $s \in S$
   $v \leftarrow V(s)$
   $V(s) \leftarrow \max_a \sum_{s'} P_{ss'}^a \left[R_{ss'}^a + \gamma V(s') \right]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   Until $\Delta < \theta$ (a small positive number)

3. Output a deterministic policy, π, such that :
   $\pi(s) = \arg\max_a \sum_{s'} P_{ss'}^a \left[R_{ss'}^a + \gamma V(s') \right]$

2.2 Description de l'environnement

Voici la description de toutes les variables à initialiser pour utiliser les algorithmes de programmation dynamique.

1. **NbStates :** Le nombre d'états (une constante).
2. **NbActions :** Le nombre d'actions possibles (une constante).
3. **TransProb :** Matrice des probabilités de transition entre les états : $TransProb(a)(i,j)$ représente la probabilité de passer de l'état i à l'état j en effectuant l'action a.
4. **Rewards** Matrice des récompenses : $Rewards(a)(i,j)$ représente la récompense obtenue en passant de l'état i à l'état j en ayant effectué l'action a.
5. **Gamma :** La variable γ utilisée dans les algorithmes. Elle doit être comprise en tre 0 et 1.
6. **Prob :** La politique initiale rentrée par le programmeur. Cette matrice n'est utile que dans l'algorithme d'évaluation d'une politique. $Prob(a,s)$ représente la probabilitée de choisir l'action a et d'aller dans l'état s.
7. **Actions_States :** Matrice des actions possibles : $Actions\_States(s,a) = 1$ si l'action a est possible dans l'état s et $Actions\_States(s,a) = 0$ si l'action est impossible.
8. **Grid :** Variable spécifique aux exemples se basant sur des grilles. Si $Grid = 1$ la grille est du type de demo_grid (deux cases de sortie de la grille) et si $Grid = 2$ la grille est du type de demo_grid2.
9. **Taille_Grille :** Variable qui réprésente la longueur de la grille (qui est sensée être carrée).

2.3 Quelques remarques sur l'implémentation des algorithmes

2.3.1 Les Macros

Boucles Infinies Après certains tests sur les algorithmes il s'est avéré que selon la politique initiale il pouvait y avoir une boucle infinie. Ainsi pour L'**Evaluation de Politique** et pour **Itération de la politique** nous avons rajouter un compteur qui permet de limiter le nombre d'itérations à un nombre fixé à l'avance. En cas de dépassement de ce compteur, l'algorithme est arrêté et l'utilisateur est averti par un message spécifique.

Valeurs Intermédiaires L'un des problèmes des algorithmes étaient qu'au moment du calcul des $V(s)$ par une formule de récurrence, la matrice V(s) n'était pas sauvegardé, ce qui entrainait des erreurs de calculs. En effet la formule est basée sur une récurrence entre les $V(s)$ de l'itération précédente pour calculer la nouvelle matrice, donc avant chaque itération on sauvegarde la matrice V sous une matrice V_old et on remplace V par V_old dans la formule. Par exemple :

$$S1 = Prob(a,s) * (TransProb(a)(s,:) * ((Rewards(a)(s,:))' + Gamma * (V(:,1))) \tag{2.9}$$

devient :

$$S1 = Prob(a,s) * (TransProb(a)(s,:) * ((Rewards(a)(s,:))' + Gamma * (V_{old}(:,1))) \tag{2.10}$$

Procédure de vérification des paramètres Cette procédure permet de vérifier les paramètres d'un algorithme afin d'éviter des boucles infinis ou des erreurs d'index. Les vérifications effectuées sont :

- Vérification des types attendus : Matrices, Listes, boolean
- Vérification des valeurs : en particulier que $0 < \gamma \leq 1$
- Vérification des tailles des matrices : Cohérence entre le nombre d'actions, le nombre d'états et la taille des différentes matrices.

Conversion d'une liste en matrice Pour afficher les valeurs des états plus simplement, nous avons développé une petite macro qui permet de transformer une liste en matrice. En effet pour les exemples de grilles, nous devions transformer la matrice V ainsi :

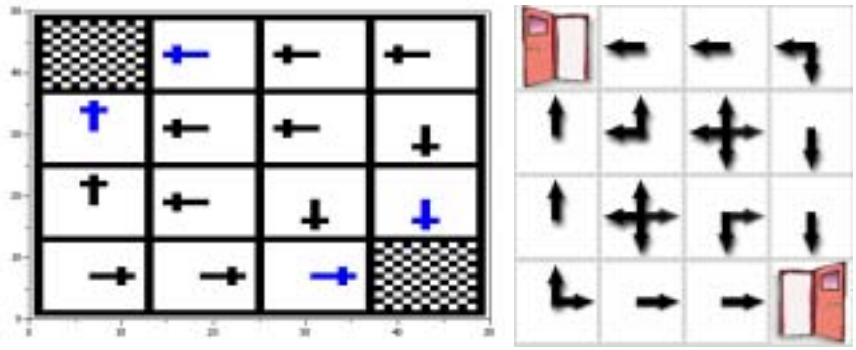
$$V\,(s_1 \cdots s_{NbrEtats}) \rightarrow V \begin{pmatrix} s_1 & \cdots & s_n \\ s_{n+1} & \cdots & s_{n+k} \\ \vdots & \ddots & \vdots \\ s_{n+k} & \cdots & s_{NbrEtats} \end{pmatrix} \tag{2.11}$$

Cette procédure s'apelle : **list2matrix**. Elle prend en argument une liste donc la taille est un carré parfait, et renvoie une matrice carrée selon la formule :

$$Ouput(i,j) = Input((i-1) * WIDTH + j) \tag{2.12}$$

Cette macro nous permet de faire l'affichage en 3d des valeurs des états.

Modification du calcul des politiques Jusqu'à présent, pour chaque état $s \in S$ on associait un nombre $\pi(s)$ indiquant l'action à effectuer dans cet état s sous la politique π. Cela est bien suffisant pour la simulation, car il suffit de connaitre une politique optimale et de l'appliquer. Cependant, quand on se place dans un cadre pédagogique, il est intéressant de montrer non pas une mais toutes les politiques optimales. Cela permet notamment de bien visualiser les différentes solutions optimales. Par exemple, nous pouvons voir sur la figure 2.1, qu'avec un problème simple où il faut trouver le plus court chemin pour sortir de la grille, il n'existe non pas une politique optimale, mais plusieurs.



(a) Une seule politique optimale

(b) Toutes les politiques optimales. L'agent a donc le choix.

FIG. 2.1 – Comparaison entre les 2 manières de représenter une politique : (a) par un scalaire (b)par un vecteur

Cela a bien entendu des répercussions importantes sur le codage des algorithmes. Ces modifications peuvent se résumer en 2 points essentiels :

1. Quand on veut s'adresser à la politique à suivre pour un état donné, il faut accéder à un vecteur et non à un scalaire.
2. Il faut distinguer dans les itérations de l'algorithme le cas où l'on obtient une valeur strictement supérieure, et le cas où l'on est égal. Dans le premier cas, il faudra recréer un vecteur complet pour décrire les politiques possible pour cet état, alors que dans le second cas, il faut compléter le vecteur existant sans perdre l'information déjà présente.

Précisemment, nous avons pour chaque état s un vecteur de booléens où chaque élément à *true* indique une action optimale pour cet état, et ceux à *false* indique une action non optimale.

Optimisation des calculs Le coeur des algorithmes est basé sur une formule similaire à celle-ci :

$$\sum_{s'} P_{ss'}^{\pi(s)}[R_{ss'}^{\pi(s)} + \gamma V(s')] \tag{2.13}$$

La première idée est donc d'utiliser la fonction **sum**, cependant on pourra remarquer que le calcul est une multiplication d'une ligne avec une colonne puis l'addition de tous les termes. Ceci correspond en réalité à une multiplication de matrice, qui a priori sera mieux optimisée sous scilab. La formule type devient donc :

$$P_{s,:}^{a} * (R_{s,:}^{a} + \gamma * (V_{:,1})) \tag{2.14}$$

Historique des Valeurs Afin d'améliorer la compréhension du mode d'apprentissage de l'agent, les résultats intermédiaires, obtenues au cours de l'apprentissage, des fonctions d'état $V(s)$ et des politiques successives π ont été mémorisées.

2.3.2 Exemples

Grid2 Nous avons implémenté en plus des exemples existants l'exemple de la grille situé dans le livre sur le Reinforcement Learning, chapitre 3. Nous nous sommes basé sur le premier exemple de grille comportant deux sorties.
Cet exemple est intéressant par rapport à l'autre grille car ce n'est pas un problème épisodique, ce qui veux dire qu'il n'a pas de fin, contrairement au problème 1 qui s'arrête dès qu'il atteind la sortie.

2.3.3 Remaniements de la présentation des résultats

Nous avons développé l'interface des exemples de la ToolBox. Pour cela, une page d'accueil à été écrite en Tcl. Elle permet de lancer les différents exemples et de charger les macros. Le fichier *RLtoolbox.sce* permet donc de lancer le fichier *RLtoolbox.tcl* qui se trouve lui aussi dans la racine de la toolbox. Ce dernier est le fichier générant la nouvelle interface. Les exemples utilisent différentes fonctions d'affichage de résultats qui ont été écrites et ajoutés dans le répertoire *Rltoolbox/METHODS/DP/Functions/Display*.

Fonctions d'affichage des résultats

Les fonctions suivantes fonctionnent pour des exemples de grilles carrées :

- **Policy_Input.sce :** cet exécutable scilab crée une fenêtre permettant de modifier et/ou vérifier la politique d'un agent sur une grille carrée de côté *Taille_Grille* et pouvant effectuer les quatres actions suivantes : *aller en haut, à gauche , en bas et à droite*. La politique doit tout de même être initialisée dans le fichier décrivant l'environnement en étant appelée *Prob* et avoir la dimension suivante *(4,NbreEtats)* (le 4 correspond aux 4 actions possibles).

- **Save_Prob.sce :** cet exécutable enregistre les nouvelles valeurs de politique entrée dans la fenêtre générée par l'exécutable précédent.

- **Affiche_Pi.sce :** cet exécutable affiche la politique π dans la fenêtre des exemples. Elle se base sur la valeur courante de la matrice *Pi* dans Scilab.

- **hist_Pi.sce :** cet exécutable permet de modifier l'affichage de la politique pour avoir celui de la politique à l'itération k avec k donné par la position de la barre de défilement dont le chemin tk doit être le suivant *.rltoolbox.f.echelle*.

- Affiche_V.sce : cet exécutable affiche la valeur des différents états V dans la fenêtre des exemples. Elle se base sur la valeur courante du vecteur V dans Scilab.

- hist_V.sce : cet exécutable permet de modifier l'affichage de la valeur des différents états pour avoir celui de la politique à l'itération k avec k donné par la position de la barre de défilement dont le chemin `tk` doit être le suivant *.rltoolbox.f.echelle*.

Nouvelle organisation des exemples

Dans un but pédagogique nous avons partitionné les exemples de grille en trois. En effet, il est désormais possible de choisir quel algorithme on veut utiliser sur l'environnement défini. Afin de permettre cela, trois différents exécutables scilab ont été écrits. Ils appelent l'algorithme qui leur est propre et s'occupent de l'affichage des résultats en conséquence. Voici le principe du nom de ces exécutables :

- **Nom_demo_Eval_Policy :** se charge d'évaluer une politique donnée et affiche les valeurs de V suivant l'itération choisie par une barre de défilement.
- **Nom_demo_Iter_Eval :** trouve une politique optimale en utilisation l'algorithme *Value_Iteration* et permet de visualiser l'évolution simultanée de V et de π'
- **Nom_demo_Iter_Improv :** trouve une politique optimale en utilisant l'algorithme *Iter_Policy_Improv* et permet de voir la politique finale ainsi que l'évolution des valeurs de V

Chapitre 3

Monte Carlo

3.1 Introduction

3.1.1 Généralités

Les méthodes de Monte Carlo apprennent à partir d'un ensemble complet de retours provenant de tâches épisodiques seulement. Une autre caractéristique de leur apprentissage est qu'il se base directement sur l'expérience. On distingue deux types d'expériences :

- **Simulée :** Aucun besoin d'une connaissance complète du modèle
- **On-line :** Pas de modèle nécessaire et obtention assurée de l'optimalité

3.1.2 Différences avec les méthodes de programmation dynamique

Tout d'abord, à partir de la partie précédente on remarque aisément qu'une première différence est que la modélisation de l'environnement ne doit pas nécessairement être complète pour les méthodes de Monte Carlo contrairement à celles de programmation dynamique. Par contre, les méthodes de Monte Carlo ne sont pas faites pour les tâches non épisodiques et ne font pas de *bootstrap*.

Enfin, un avantage des méthodes de Monte Carlo tient dans le fait que le temps nécessaire à l'estimation d'un état ne dépend pas du nombre total d'états (contrairement aux méthodes de programmation dynamique).

3.2 Evaluation de Politique

Les méthodes de Monte Carlo évaluent une politique pour un état s en se basant sur la moyenne des retours observés au cours de différents épisodes pour ce même état. On distingue deux approches différentes de calcul des moyennes qui convergent assymptotiquement :

- **Every-Visit MC :** la moyenne des retours s'effectue en tenant compte de toutes les fois où l'état s est visité dans un épisode.

- **First-Visit MC :** la moyenne des retours s'effectue en tenant compte que de la première fois qu'un état s est visité dans un épisode.

Par la suite on utilisera la deuxième approche. L'algorithme d'évaluation d'une politique se trouve en 3.1 (page 22).

ALGO. 3.1 Algorithme *First Visit Monte Carlo* d'évaluation d'une politique

1. Initialize :
   $\pi \leftarrow$ policy to be evaluated
   $V \leftarrow$ an arbitrary state-value function
   $Returns(s) \leftarrow$ an empty list, for all $s \in S$

2. Repeat Forever
   (a) Generate an episode using π
   (b) For each state s appearing in the episode :
   $R \leftarrow$ return following the first occurence of s
   Append R to $Returns(s)$
   $V(s) \leftarrow average(Returns(s))$

3.3 Estimation de la valeur d'une action

Sans un modèle les valeurs d'états sont insuffisantes pour trouver une politique optimum, le but premier des méthodes de Monte Carlo est donc d'estimer $Q^{\pi}(s,a)$. Pour cela, on fait la moyenne des retours partant de l'état s et de l'action a en suivant le politique π. La convergence est assurée si chaque paire $etat-action$ est visitée. Pour cela on peut faire l'hypothèse des *départs d'exploration*, c'est à dire qu'on considère que chaque couple $etat-action$ a une probabilité différente de 0 d'être la paire de départ d'un épisode.

3.4 Contrôle Monte Carlo

Il consiste en une évaluation de politique utilisant les méthodes de Monte Carlo suivie d'une amélioration de politique. Cela implique que les hypothèses de *départs d'exploration* et de *nombre infinis d'épisodes* sont vérifiées. Le deuxième point peut être résolu en effectuant une mise à jour répondant à un niveau de performance donné et en alternant l'évaluation et l'amélioration par épisode. On aboutit donc a l'algorithme se trouvant en 3.2.

ALGO. 3.2 Algorithme Contrôle Monte Carlo avec départs d'exploration

1. Initialize, for all $s \in S, a \in A(s)$:
   $Q(s,a) \leftarrow$ arbitrary
   $\pi(s) \leftarrow$ arbitrary
   $Returns(s) \leftarrow$ empty list

2. Repeat Forever
   (a) Generate an episode using exploring starts and π
   (b) For each pair s,a appearing in the episode :
   $R \leftarrow$ return following the first occurence of s,a
   Append R to $Returns(s,a)$
   $Q(s,a) \leftarrow average(Returns(s,a))$
   (c) For each s in the episode :
   $\pi(s) \leftarrow argmax_a Q(s,a)$

3.5 On-Policy

Cette méthode permet d'éliminer le besoin de vérifier l'hypothèse des *départs d'exploration* et d'apprendre au sujet de la politique en cours d'exécution. Tout d'abord, pour se passer de l'hy-

pothèse des *départs d'exploration* il faut utiliser une *politique douce*[1], c'est à dire une politique pour laquelle $\pi(s,a) > 0$ quelques soient s et a. Cette méthode converge vers la meilleure *politique douce*. On peut noter la propriété suivante : une politique $\epsilon-$gloutonne respectant Q^π est une amélioration de n'importe laquelle des politiques $\epsilon-$douce pour la politique π.

L'algorithme est donné en 3.3.

ALGO. 3.3 Algorithme *On-Policy Monte Carlo*

1. Initialize, for all $s \in S, a \in A(s)$:
   $Q(s,a) \leftarrow$ arbitrary
   $\pi(s) \leftarrow$ arbitrary $\epsilon-$soft policy
   $Returns(s) \leftarrow$ empty list

2. Repeat Forever
   (a) Generate an episode using exploring starts and π
   (b) For each pair s,a appearing in the episode :
   $R \leftarrow$ return following the first occurence of s,a
   Append R to $Returns(s,a)$
   $Q(s,a) \leftarrow average(Returns(s,a))$
   (c) For each s in the episode :
   $a* \leftarrow argmax_a Q(s,a)$
   For all $a \in A(s)$:

$$\pi(s,a) \leftarrow \begin{cases} \frac{1-\epsilon+\epsilon}{|A(s)|} & if\ a = a* \\ \frac{\epsilon}{|A(s)|} & if\ a \neq a* \end{cases}$$

3.6 Off-Policy

Pour cette méthode on définit deux types de politiques :

- **Politique comportementale :** c'est la politique qui va être suivie pour la génération de l'épisode
- **Politique estimée :** c'est la politique qui est en cours d'apprentissage

Pour pouvoir utiliser ces deux types de politique il faut donc pouvoir estimer une politique π tout en suivant une politique π'. En supposant qu'on ait n_s retours, $R_i(s)$, à partir de l'état s, chacun ayant la probabilité $p_i(s)$ d'être généré par π et la probabilité $p'_i(s)$ d'être généré par π', on peut donc estimer :

$$V^\pi(s) = \frac{\sum_{i=1}^{n_s} \frac{p_i(s)}{p'_i(s)} R_i(s)}{\sum_{i=1}^{n_s} \frac{p_i(s)}{p'_i(s)}}$$

et on a :

$$\frac{p_i(s)}{p'_i(s)} = \prod_{k=t}^{T_i(s)-1} \frac{\pi(s_k,a_k)}{\pi'(s_k,a_k)}$$

A partir de tout cela on aboutit donc à l'algorithme 3.4 (page 24) qui permet donc d'obtenir une politique déterministe tout en parcourant tous les paires $etat - action$ grâce à une politique comportementale $\epsilon-$douce.

[1] Soft Policy

ALGO. 3.4 Algorithme *Off-Policy Monte Carlo*

1. Initialize, for all $s \in S, a \in A(s)$:
   $Q(s,a) \leftarrow$ arbitrary
   $\pi \leftarrow$ an arbitrary deterministic policy
   $N(s,a) \leftarrow 0$: Numerator
   $D(s,a) \leftarrow 0$: and Denominator of Q(s,a)

2. Repeat Forever
   (a) Select a policy π' and use it to generate an episode :
   $s_0, a_0, r_1, s_1, a_1, r_2, ..., s_{T-1}, r_T, s_T$
   (b) $\tau \leftarrow$ latest time at which $a_\tau \neq \pi(s_\tau)$
   (c) For each pair s, a appearing in the episode after τ :
   $t \leftarrow$ the time of the first occurence (after τ) of s, a
   $w \leftarrow \prod_{k=t+1}^{T-1} \frac{1}{\pi'(s_k, a_k)}$
   $N(s,a) \leftarrow N(s,a) + wR_t$
   $D(s,a) \leftarrow D(s,a) + w$
   $Q(s,a) \leftarrow \frac{N(s,a)}{D(s,a)}$
   (d) For each $s \in S$:
   $\pi(s) \leftarrow argmax_a Q(s,a)$

Chapitre 4

Temporal Difference

4.1 Théorie

4.1.1 Introduction

Les méthodes de *Temporal Difference* combinent des idées utilisées dans les méthodes de *programmation dynamique* et de *MonteCarlo*. En effet, la mise à jour des estimations est basée sur les estimations des états successeurs, ceci est appelé *bootstraps* comme pour les méthodes de *programmation dynamique*. Par contre, on reprend les idées développées pour les méthodes de *MonteCarlo* en effectuant un apprentissage par l'expérience (échantilonnage) sans le modèle de la dynamique de l'environnement.

Cela nous amène donc en résumé à la liste d'avantages suivante pour les méthodes de *Temporal Difference* :

- Aucun modèle de l'environnement est nécessaire, l'expérience suffit.
- L'apprentissage se fait avant de connaitre la récompense finale voire sans la connaitre.
- Les méthodes convergent.

4.1.2 TD Prediction

Le but de l'algorithme est de calculer les fonctions valeurs d'états V^π. Le pseudo code de cet algorithme se trouve en 4.1 (page 25).

Algo. 4.1 Algorithme *TD(0)* d'évaluation d'une politique

1. Initialize $V(s)$ arbitrarily, π to the policy to be evaluated :

2. Repeat (for each episode) :
   Initialize s
   Repeat (for each step of episode) :
   $a \leftarrow$ action given by π for s
   Take action a ; observe reward, r, and next state, s'
   $V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$
   $s \leftarrow s'$
   until s is terminal

Optimalité de TD(0)

Pour cela, on va introduire une nouvelle notion : *Batch Updating*. Cela consiste à effectuer un

apprentissage sur un groupe fini de données, par exemple en effectuant un apprentissage sur 10 épisodes jusqu'à la convergence. Les calculs sont mis à jour mais seulement après chaque passage par les données.

Pour toute tâche prédictive de Markov finie, avec un *Batch Updating*, TD(0) converge pour un α suffisament petit, tout comme $\alpha-$constante *MonteCarlo*. Cependant on peut noter que les deux algorithmes aboutissent à des réponses optimales différentes.

4.1.3 Sarsa : On-Policy TD Control

L'algorithme se trouvant en 4.2 (page 26) permet d'obtenir une politique optimale en se basant sur les $Q(s,a)$.

Algo. 4.2 Algorithme *Sarsa*

1. Initialize $Q(s,a)$ arbitrarily :

2. Repeat (for each episode) :
   Initialize s
   Choose a from s using policy derived from Q (e.g., $\epsilon-$greedy)
   Repeat (for each step of episode) :
   Take action a, observer r, s'
   Choose a' from s' using policy derived from Q (e.g., $\epsilon-$greedy)
   $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$
   $s \leftarrow s'$; $a \leftarrow a'$
   until s is terminal

4.1.4 Q-learning : Off-Policy TD Control

Pour chaque étape du *Q-learning* on a :

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)]$$

L'algorithme qui en decoule se trouve en 4.3 (page 26).

Algo. 4.3 Algorithme *Q-learning*

1. Initialize $Q(s,a)$ arbitrarily :

2. Repeat (for each episode) :
   Initialize s
   Repeat (for each step of episode) :
   Choose a from s using policy derived from Q (e.g., $\epsilon-$greedy)
   Take action a, observer r, s'
   $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
   $s \leftarrow s'$;
   until s is terminal

4.1.5 Méthodes Acteur-Critique

Les méthodes acteur-critique sont des façons différentes de modéliser le problème de l'apprentissage par renforcement, en particulier pour les méthodes de *Temporal Difference*. En pratique la politique π est définie comme l'**acteur** et la fonction valeur comme la **critique**. En effet, la

politique est celle qui va décider des actions à effectuer, alors que la fonction valeur est là pour critiquer (dans le sens positif ou négatif) les actions entreprises.

L'erreur TD est utilisée pour évaluer (critiquer) les actions :

$$\delta_t = t_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{4.1}$$

Si les actions sont déterminées selon la méthode softmax de Gibbs (cf. rubrique 1.3.1 page 11) par les préférences $p(s,a)$ comme suit :

$$\pi_t(s,a) = Pra_t = a|s_t = s = \frac{e^{p(s,a)}}{\sum_b e^{p(s,a)}} \tag{4.2}$$

On peut alors mettre à jour les préférences en fonction de la critique déterminée par l'erreur TD (équation 4.1) comme suit :

$$p(s_t,a_t) \leftarrow p(s_t,a_t) + \beta\delta_t \tag{4.3}$$

Ces méthodes n'ont pas été très utilisées depuis quelques années, les efforts se sont plutôt portés sur des méthodes qui détermine la politique exclusivement des valeurs estimées(par exemple Sarsa et Q-learning). Cependant ces méthodes conservent des interêts non négligeables :

1. **Calculs informatiques moins couteux :** Dans les exemples à nombre infinis d'actions possibles il n'est pas forcément nécessaire de tester chaque action pour décider laquelle prendre, à condition que la politique soit bien triée.
2. **Apprentissage sur des politiques stochastiques :** Avantage très interessant dans les modèles qui ne sont pas de Markov.
3. **Application aux modèles psychologiques et biologiques :** Cette approche permet de mieux coller aux simulations biologiques car l'acteur est bien séparé.

4.1.6 R-learning : Off-Policy TD Control

L'approche R-learning est adaptée aux expériences infinies, en effet elle ne nécessite pas de découpage en épisodes finis. Ainsi le but recherché est de maximiser la récompense moyenne et non la récomprense finale.

On peut calculer la récompense moyenne espérée par étape sous une politique π comme suit :

$$\rho^\pi = \lim_{n\to\infty} \frac{1}{n} \sum_{t=1}^{n} E_\pi r_t \tag{4.4}$$

Ce qui nous donne les valeurs d'états relatives à ρ^π suivantes :

$$\tilde{V}^\pi(s) = \sum_{k=1}^{\infty} E_\pi\{r_{t+k} - \rho^\pi | s_t = s\} \tag{4.5}$$

et les valeurs d'*actions-états* suivantes :

$$\tilde{Q}^\pi(s,a) = \sum_{k=1}^{\infty} E_\pi\{r_{t+k} - \rho^\pi | s_t = s, a_t = a\} \tag{4.6}$$

L'algorithme qui en découle se trouve en 4.4 (page 28).

4.2 Description de l'environnement

Afin de réaliser un modèle d'environnement, nous avons besoin de plusieurs informations. Tout d'abord, nous avons besoin de certaines variables qui permettent de paramétrer les différents algorithmes. A cela s'ajoute un certain nombre de fonctions qui servent à décrire le comportement de l'environnement.

ALGO. 4.4 Algorithme *R-learning*

1. Initialize ρ and $Q(s,a)$, for all s,a, arbitrarily :

2. Repeat forever :
   $s \leftarrow$ current state
   Choose action a in s using behavior policy (e.g., $\epsilon-$greedy)
   Take action a, observer r, s'
   $Q(s,a) \leftarrow Q(s,a) + \alpha[r - \rho + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
   If $Q(s,a) = max_a Q(s,a)$, then :
   $\rho \leftarrow \rho + \beta[r - \rho + \gamma \max_{a'} Q(s',a') - \max_a Q(s,a)]$

4.2.1 Variables et fonctions décrivant un environnement

Regardons tout d'abord les variables nécessaires pour paramétrer chaque algorithme :

1. **Alpha :** Paramètre permettant de faire varier la vitesse de converge et la précision de la méthode. Il est utilisé par les algorithmes 4.1, 4.2, 4.3 et 4.4.
2. **Beta :** Paramètre similaire à Alpha, mais il est utilisé uniquement avec le *R-Learning* 4.4.
3. **Epsilon :** Paramètre utilisé par les algorithmes 4.2, 4.3 et 4.4 permettant de choisir une action suivant une politique ϵ-greedy. Le choix de l'action est basé sur les valeurs de Q.
4. **Gamma :** Paramètre permettant de fixer la "mémoire" de l'agent. Ses valeurs sont comprises entre 0 et 1, avec 1 qui correspond à une mémoire infinie et 0 qui correspond à aucune mémoire. Pour une tâche épisodique on utilise normalement $\gamma = 1$.
5. **NbStates :** Nombre d'états possibles dans notre environnement.
6. **NbActions :** Nombre maximum d'actions associées à un état.
7. **ActionsSates :** Matrice de booléens indiquant quelles actions sont possibles pour chaque état. On retrouve les états en ligne, et les actions en colonne. Une action possible dans un état se traduira par un 1 à la bonne case, dans le cas contraire on aura un 0. Cette représentation suppose qu'il est possible de numéroter les états et les actions.
8. **NbEpisodes :** Les algorithmes 4.1, 4.2 et 4.3 utilisent tous les trois un nombre d'épisodes prédéfini pour déterminer quand ils doivent s'arrêter. Ce nombre d'itérations est règlé par ce paramètre. Quant à l'algorithme 4.4, théoriquement il boucle à l'infinie, mais pratiquement nous avons décidé de le limiter à un certain nombre d'itérations qui sera fixé par ce paramètre. Dans tous les cas, ce paramètre est demandé à l'utilisateur par le biais d'une boite de dialogue.

Les fonctions suivantes permettent de décrire un environnement et sont appelés par tous les algorithmes.

1. **TD_Mdl_first_state :** Cette fonction donne le premier état d'un épisode.
2. **TD_Mdl_terminal_state :** Cette fonction prend en paramètre le numéro d'un état et renvoie 1 si l'état en question est un état terminal, et 0 sinon.
3. **TD_Mdl_Policy :** Cette fonction prend en paramètre le numéro d'un état, et donne l'action choisie à partir de cet état. Le but de cette fonction est de décrire une politique qui sera évaluer avec l'algorithme TD0 (l'algorithme 4.1).
4. **TD_Mdl_nextstate_reward :** Cette fonction prend en paramètre un état et une action, et elle donne l'état futur s' ainsi que la récompense obtenue.

4.2.2 Arborescence des fichiers

De manière à ce que toutes ces fonctions soient simples d'utilisation une interface graphique a été développée. Le fonctionnement de cette interface demande de bien organiser ses fichiers, et de respecter certaines conventions de nomage. Dans la suite, tous les chemins seront des chemins relatif à la racine de la toolbox.

Le dossier METHODS/TD/ est la racine de toute la partie Temporal Difference de la toolbox.

Le dossier METHODS/TD/Doc/ contient la documentation relative à cette partie de la toolbox.

Le dossier METHODS/TD/Functions/ contient l'implémentation des algorithmes présentés dans le livre de Sutton et Barton. Il y a un fichier par algorithme. A cela s'ajoute un fichier TD_Functions.sci pour des fonctions utilitaires. On y trouve en particulier des fonctions comme le choix d'une action à partir d'un état en suivant une politique ϵ-greedy basée sur Q. Enfin, il y a un dernier fichier, TD_load_functions.sce qui est utilisé par l'interface graphique de manière à charger uniquement les algorithmes sans exécuter d'exemple.

Le dossier METHODS/TD/Examples/ contient differents dossiers correspondants chacun à un exemple différent. Par exemple, nous avons un dossier RandomWalk pour la marche aléatoire, et le dossier WindyGrid pour l'exemple de grille comprenant du vent. Dans chacun de ces sous dossiers, nous retrouvons obligatoirement le fichier Mdl_exec.sce. C'est ce fichier qui est appelé par l'interface graphique, et c'est à partir de ce fichier que tout le reste est exécuté. Un autre fichier courant est Mdl_functions qui contient les fonctions décrivant un environnement. Les autres fichiers que l'on peut trouver dans ces dossier servent soient à l'interface graphique, soit à définir d'autres comportements pour un même modèle (généralement des fichiers du type Mdl_loadExemple.mdl.

4.3 Exemples d'applications

4.3.1 Marche Aléatoire (Random Walk)

Cet exemple est directement tiré du livre de Sutton (Chapitre 6.2). Celui-ci est un modèle de Markov qui va générer des marches aléatoires. Ce modèle est représenté sur la figure 4.1. L'état initial est au centre, ensuite à chaque étape on peut aller aléatoirement à gauche ou à droite, jusqu'au états terminaux. La seule récompense est quand on termine l'épisode par la droite.

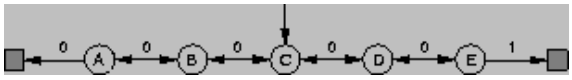


FIG. 4.1 – Modèle de Markov pour la marche aléatoire. Les états carrés sont terminaux

Le but de cet exemple est de tester l'algorithme de prédiction (cf. algorithme 4.1 page 25). En effet une politique aléatoire est évaluée à chaque itération, et le retour de l'algorithme est la fonction valeur-état $V^{\pi}(s)$.

Les valeurs théoriques de cet exemple sont facilement identifiables et valent pour chaque état de A à E : $\frac{1}{6}$, $\frac{2}{6}$, $\frac{3}{6}$, $\frac{4}{6}$, et $\frac{5}{6}$. On peux voir les résultats obtenus par l'algorithme sur la figure 4.2.

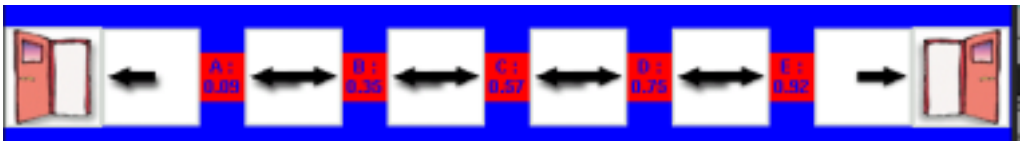


FIG. 4.2 – Valeur des états visualisées dans l'interface obtenues par l'algorithme TD0 sur l'exemple de marche aléatoire

Sur la figure 4.2 on observe les évaluations de l'algorithme TD(0). Sur la première figure on voit l'évolution des valeurs d'états en fonction du nombres d'épisodes simulés, on remarquera que l'algorithme semble converger à partir de quelques centaines d'épisodes. Ensuite on visualise la comparaison de l'algorithme TD prediction et le MonteCarlo-α-constante, on observe que l'algorithme de MonteCarlo donne des erreurs absolues environ deux fois plus importantes pour un même nombre d'itérations identiques(ici 500).

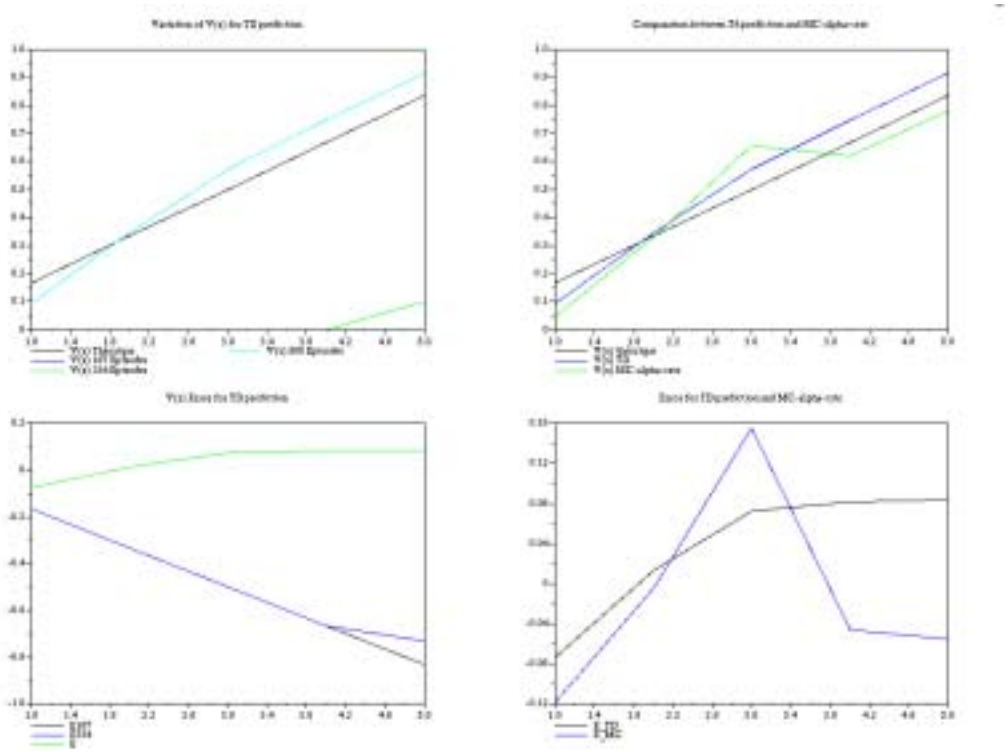


FIG. 4.3 – Résultats obtenus par l'algorithme TD0 sur l'exemple de marche aléatoire

4.3.2 Windy Grid World

Windy Grid World Cet exemple reprend celui d'une grille comportant une entrée et un but à atteindre, cependant on va rajouter du vent vertical. Lors d'un déplacement dans une case comportant du vent on va tenir compte de celui-ci afin de déterminer la case d'arrivée. Par exemple si l'action est droite et qu'il y a un vent de +1 vers le haut, la case arrivée sera en haut à droite du départ.

Le but de l'exemple est de démontrer l'efficacité de l'algorithme Sarsa afin de trouver le chemin optimal permettant d'atteindre le but en minimisant le nombre d'étapes.

Après quelques tests on obtient la solution optimale visible sur la figure 4.4 au bout de 1000 itérations (avec $\epsilon = 0.1$ et $\alpha = 0.1$). Grace à cet exemple simple nous avons pût vérifier que l'algorithme codé dans la boite à outil se comporte correctement.

Cliff Walking Le deuxième exemple, directement tiré du livre, que nous avons implémenté est le "Cliff Walking" . Celui-ci est également basé sur une grille comportant une entrée et une sortie, cependant sur la dernière ligne un trou est rajouté. Celui-ci renvoie au point de départ si on tombe

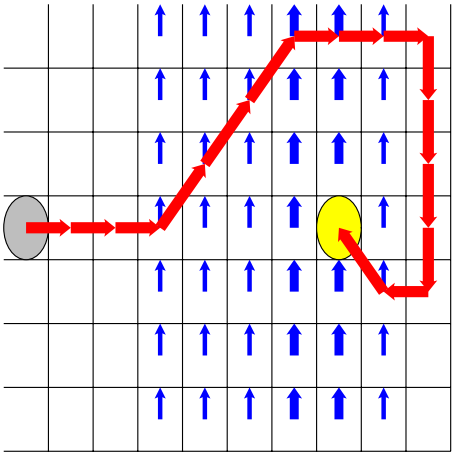
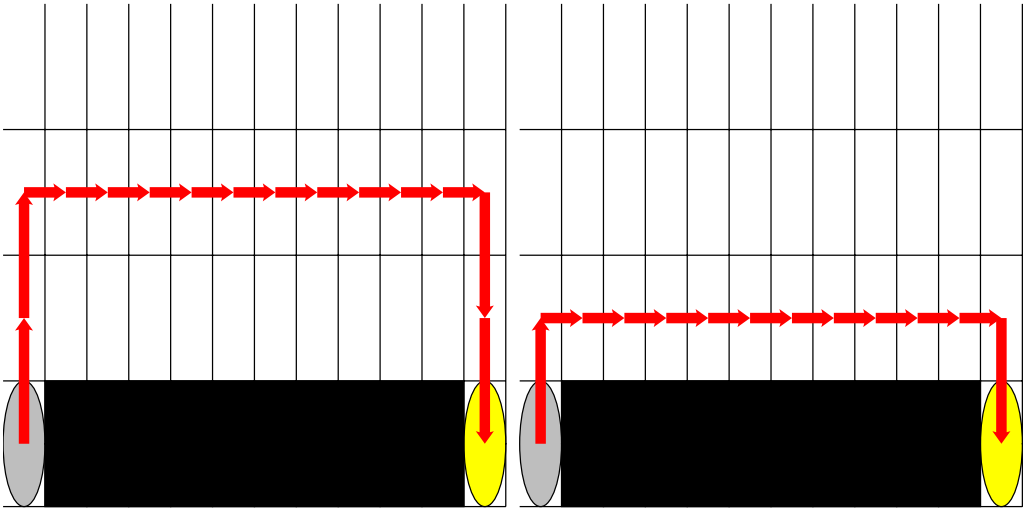


FIG. 4.4 – Solution Optimale de la grille ventée obtenue après 1000 épisodes sur l'algorithme Sarsa

dedans et avec une récompense de -100. On voit sur la figure 4.5(b) que l'algorithme Q-learning converge très rapidement en 300 itérations, alors que l'algorithme Sarsa trouve une solution sous-optimale, mais plus sure (c'est a dire plus loin de la falaise). En revanche si on augmente le nombre d'épisodes, Sarsa converge également vers la solution optimale. La figure 4.5 page 31 présente les résultats obtenus suivant l'algorithme utilisé.



(a) Solution sous optimale par TD Sarsa (500 itérations)

(b) Solution optimale par TD Q-Learning (300 itérations)

FIG. 4.5 – Comparaison entre deux algorithmes sur l'exemple du Cliff Walking

4.4 Généralisation du Windy Grid World

4.4.1 But

L'idée est de combiner les deux exemples basiques de la falaise et du vent afin d'imaginer des grilles plus évoluées. Nous voulions avoir un environnement complètement configurable afin de

pouvoir tester les algorithmes de Temporal Difference codés dans la toolbox. Nous avons donc pensé à pouvoir rajouter des falaises, des murs, du vent et des trous. Voici la description des différents éléments de la nouvelle Grille.

- Le vent : Chaque case pourra avoir un vent en X et un vent en Y, on peut ainsi modéliser tous les vents possibles.
- Les murs : Bloque tout mouvement.
- Les trous : Obstacle qui arrête le mouvement, mais on peut en ressortir au coup d'après.
- Les falaises : Dès que l'on tombe dedans on est réexpédié au départ de la grille.

Le but est d'avoir une boite à outil très modulable, par exemple il devra être simple de rajouter ses environnements spécifiques, mais également de pouvoir charger différents modèles de fonctions pour un exemple spécifique.

4.4.2 Environnement spécifique

Pour gérer ces environnements configurables, nous utilisons des fichiers modèles situés dans le dossier Models (*RLtoolbox/METHODS/TD/Examples/WindyGrid/Models*). Ces fichiers contiennent les variables d'environnement nécessaire aux grilles. On retrouve les valeurs communes à tous les examples (alpha, epsilon, ...), se référer à la rubrique 4.2.1 pour une description détaillée.
Du fait de la spécification des grilles nous avons rajouté une partie spécifique aux WindyGrid-World, dont voici les constantes communes à toutes les grilles :

- Les types de cases :
  - *NORMAL*=0 : représente une case normale
  - *CLIFF*=1 représente une case comportant une falaise
  - *WALL*=2 : représente un mur au milieu de la grille
  - *HOLE*=3 : représente un trou
  - *BORDER*=4 représente une bordure de la grille

- Les actions possibles :
  - *GAUCHE*=1
  - *BAS*=2
  - *DROITE*=3
  - *HAUT*=4

Les variables qui sont susceptibles de changer d'une grille à l'autre sont :
- *Reward_goal* : récompense pour l'atteinte d'un but
- *Reward_normal* : récompense pour un mouvement normal
- *Reward_border* : récompense pour avoir touché une bordure
- *Reward_Cliff* : récompense pour être tombé dans une falaise
- *Reward_Wall* : récompense pour avoir touché un mur
- *Reward_Hole* : récompense pour avoir rencontré un trou

- *grid_width* : Largeur de la grille
- *grid_height* : Hauteur de la grille
- *First_State* : Numéro de l'état initial.

- *Term_States* : Matrice des états terminaux, Term_states(y,x) = 1 si la case de coordonnée (x,y) est terminale, 0 sinon. Taille de la matrice : *grid_height x grid_width*
- *Wind_X et Wind_Y* : Matrice des vents, Wind_X(y,x) donne la composante en X du vent sur la case de coordonnée (x,y) (idem pour Y). Taille de la matrice : *grid_height x grid_width*

– *Obstacles* : Matrice des obstacles, Obstacles(y,x) donne le type d'obstacle se trouvant sur la case de coordonnée (x,y) (voir les types d'obstacles ci - dessus). Taille de la matrice : *grid_height x grid_width*

4.4.3 Calcul des trajectoires et détection des obstacles

Afin de gérer la nouvelle grille nous devons détecter les obstacles à chaque déplacement, afin d'en déterminer la récompense, ainsi que la case réelle d'arrivée.

Calcul de trajectoire discrète La première étape est le calcul de la position d'arrivée virtuelle dépendant du vent et de l'action initiale (trajectoire 1 sur la figure 4.7). Pour cela nous avons implémenté l'algorithme de Bresenham (algorithme 4.5) afin de déterminer la trajectoire discrète passant par le point de départ et celui d'arrivée. L'algorithme présenté ici ne fonctionne que pour une trajectoire ayant un coefficient directeur (m) entre -1 et 1, pour les droites ayant un coefficient supérieur, la boucle doit se faire sur les y et non sur les x. Enfin à l'implémentation nous avons également séparé les cas verticaux (m=∞) et horizontaux (m=0).

La procédure bresenham implémentée dans l'exemple du GridWorld nous renvoie une matrice contenant la liste des coordonnées composant la trajectoire dans le sens de parcours.

Algo. 4.5 Algorithme de Bresenham

m = $\frac{y_2-y_1}{x_2-x_1}$
y=y_1
eps=0
Pour x de x_1 à x_2
  Ajouter x,y à la trajectoire
  eps=eps+a
  Si $eps > \frac{1}{2}$
    y=y+1
    eps=eps-1
  FinSi
finPour

Les bords Après avoir calculé la position d'arrivée virtuelle, nous testons si elle est hors de la grille puis nous calculons la position réelle comme étant la projection orthogonale du point d'arrivée sur la colonne le long de la bordure (tracé 2 sur la figure 4.7). Cette méthode permet de simuler le vent qui pousse l'agent contre le mur.

Les obstacles Les obstacles sont traités à chaque nouveau mouvement, après avoir appliqué l'algorithme de Bresenham chaque point successif de la trajectoire est testé afin de savoir si on heurte un obstacle. Les cas possibles sont :

– Une case normale : On avance au point suivant, la récompense est *Reward_normal*
– Une bordure : On calcul le point d'arrivé comme précisé ci-dessus et on s'arrête , la récompense est *Reward_border*
– Une Cliff : On se positionne dans l'état initial et on arrête le parcours, la récompense est *Reward_cliff*
– Un mur : On s'arrête sur la position précédente dans la trajectoire, la récompense est *Reward_wall*
– Un trou : On s'arrête sur la position actuelle dans la trajectoire (sauf si on est dans la première position), la récompense est *Reward_hole*

FIG. 4.6 – Calcul de position suite à la collision avec une bordure

4.4.4 Editeur de modèle

Nous avons développé un éditeur de modèles spécifiques aux Windy Grid World afin de permettre de configurer aisemment un nouvel environnement (cf. figure 4.7 page 35). Celui-ci permet de créer ou de modifier un modèle de manière aisé à l'aide d'une interface graphique. On peut à l'aide de celle ci redéfinir les récompenses des différents éléments du modèle ainsi que les variables des algorithmes (alpha, beta et epsilon). On peut aussi placer à la souris sur la grille les éléments suivants : le départ, la ou les case(s) de fin et le ou les obstacle(s). Pour enlever un élément il suffit de (re)cliquer dessus sauf pour le départ qui ne peut pas être absent de la grille (attention toujours vérifier qu'il reste un point d'arrivé de défini). Pour choisir quel élément on veut mettre ou enlever il faut cliquer sur le bouton radio se trouvant devant le nom de l'élément souhaité.

Le départ est défini sur la grille par un rond gris, la ou les arrivées par des ronds jaunes, les murs par des carrés rouges avec des bandes blanches, les cliffs par des carrés noirs et les trous par des carrés noirs avec un disque blanc au milieu. Enfin, le vent peut être défini ligne par ligne et colonne par colonne. Il suffit de remplir les zones de textes se trouvant en bout de chaque ligne et de chaque colonne en indiquant un chiffre entier (représentant la force du vent). Ce dernier sera positif si on veut que le vent aille vers la zone de texte et négatif si on veut aller dans l'autre sens. Le vent sera alors affiché par des flèches bleues proportionnelles à la force du vent. Néanmoins, il est possible de définir le vent case par case, mais pour cela il faut éditer le fichier *.mdl* généré par l'interface à la main.

En résumé, cet éditeur permet de modifier les données de l'environnements suivantes :

- **Les paramètres :** α, β, ϵ
- **Les récompenses spécifiques :** Goal Reward, Normal Reward, Border Reward, Wall Reward, Cliff Reward et Hole Reward
- **Le départ et les buts :** Start et Finish
- **Les obstacles :** Wall, Cliff, Hole
- **Le vent :** spécifié suivant chaque ligne et chaque colonne

Pour créer une nouvelle grille il suffit de charger le modèle Mdl_LoadNew.mdl.

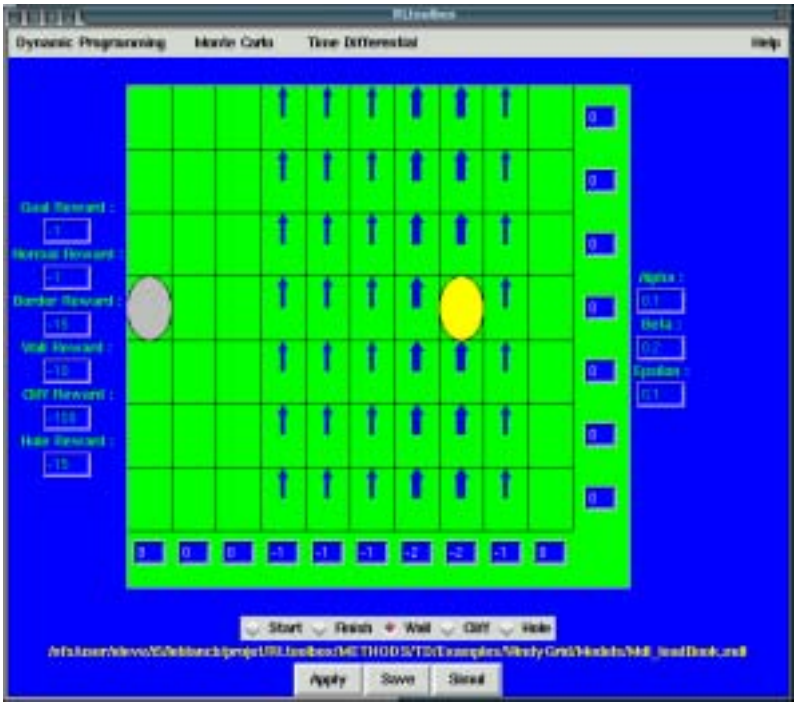


FIG. 4.7 – Editeur d'environnement pour le Windy Grid World

Il est important de noter que quand on souhaite créer une grille avec du vent, il est possible d'amener l'agent à explorer des situations desquelles il ne pourra jamais se sortir. Un cas typique est quand on met du vent vers le bas et vers la droite dans la case qui se trouve en bas à droite. Si l'agent se retrouve dans cette case, alors toutes les actions dont il dispose le ramèneront dans cette même case. Une autre possiblité est avec les murs. Si l'on dessine un cul de sac avec des murs et que l'on place un vent qui pousse vers le fond de ce cul de sac, alors il est tout à fait possible de s'y retrouver bloqué. Malheureusement ces deux exemples ne sont que des exemples et il est préférable de faire bien attention à ce que l'on fait avec le vent.

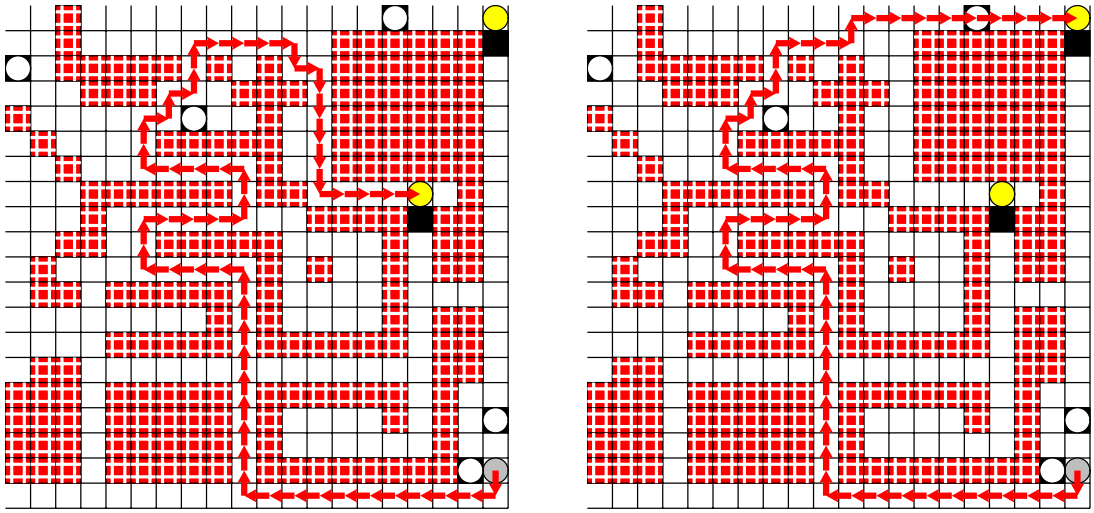
4.4.5 Exemples d'applications

Grâce à l'éditeur de grille et aux obstacles qui ont été introduit précédemment, il est possible de modéliser des environnement plus complexes. Par conséquent, nous nous sommes attachés à deux exemples particuliers : un labyrinthe avec des pièges, et un autre labyrinthe auquel on a ajouté du vent.

Dans le premier labyrinthe, nous avons testé la capacité de l'agent utilisant *Temporal Difference* à trouver le chemin optimal. Il faut savoir que ce chemin optimal n'est pas le chemin le plus direct étant donné que l'on a placé un certain nombre d'obstacles pénalisants. De plus, plusieurs arrivées sont possibles. L'agent devra donc choisir celle qui lui permet d'obtenir le chemin optimal, c'est à dire le chemin le moins pénalisant sachant que chaque déplacement et chaque obstacle a un coût qui lui est propre.

Dans l'exemple de la figure 4.8 on peut voir que la pénalité associée à un obstacle peut changer la sortie choisie et par conséquent le chemin retenu. Cela met en évidence que *les algorithmes*

utilisés sont généraux et ne tiennent compte que du but et non du moyen, en se basant sur les interactions avec l'environnement, et donc des récompenses qui en découlent. C'est pour cela qu'une modélisation adéquate du problème est primordiale pour résoudre le problème voulu. En effet, dans les deux cas nous avions une récompense de -1 pour un déplacement normal. Cependant, dans le premier cas une récompense de -5 était attribuée à un trou tandis que dans le deuxième cas nous avions seulement une récompense de -1.



(a) Labyrinthe avec $R_{trou} = -5$. L'agent choisi la sortie du bas bien que le chemin soit plus long en nombre de cases mais moins pénalisant au total.

(b) Labyrinthe avec $R_{trou} = -1$. Cette fois ci, la sortie du haut qui a un chemin plus court est atteinte car le trou pénalise désormais moins que le surplus de déplacement pour atteindre l'autre sortie.

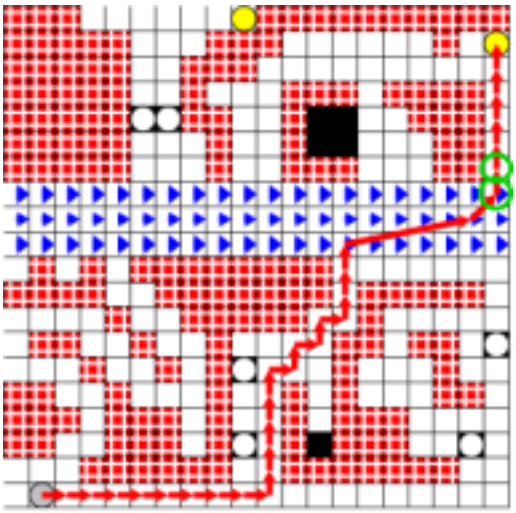
FIG. 4.8 – Labyrinthe avec obstacles

Les différentes pénalités associées à un trou pourrait s'interprêter comme l'importance des dégâts subits par l'agent quand il passe par un tel obstacle. Dans le premier cas, l'agent se méfiera plus des trous car il se rendra compte que cela lui occasionne de gros problèmes. Cependant, dans le deuxième cas, les pertes engendrées par un trou sont minimes, et l'agent pourra plus facilement se permettre de traverser une case de ce type pour atteindre son but.

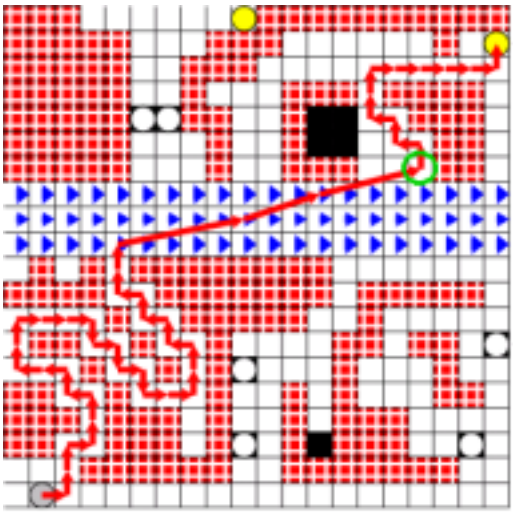
Avec le deuxième exemple de labyrinthe, les mêmes obstacles sont utilisés, mais la force du vent est ajoutée. Cela va nous permettre de mesurer son impact sur le chemin emprunté, et notamment le fait que cela peut obliger l'agent à se heurter sur un bord de la grille ou sur un mur. En effet, la particularité de cette grille est qu'il n'existe aucun chemin permettant d'aller du point de départ au point d'arrivée sans rencontrer de tels obstacles. Cela vient du fait que la force du vent est assez forte est bien répartie.

Nous avons fait varier les récompenses associées aux murs et aux bords afin de voir comment s'adapte l'agent dans ces situations là. Les résultats obtenus sont présentés dans la figure 4.9 page 37.

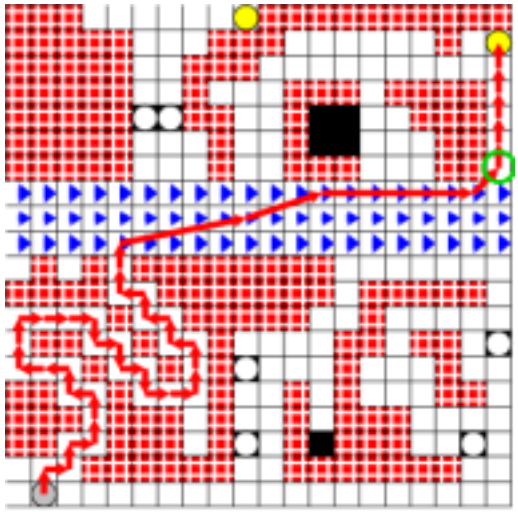
Ici encore, l'agent a à chaque fois choisi le meilleur compromis entre la distance parcourue et les obstacles rencontrés suivant les pénalités engendrés par ces derniers. Nous observons donc que l'agent sait s'adapter en fonction de l'environnement dans lequel il évolue.



(a) Labyrinthe avec $R_{bord} = -10$ et $R_{mur} = -25$. L'agent préfère se heurter deux fois sur le bord tout en empruntant le chemin le plus direct.



(b) Labyrinthe avec $R_{bord} = -25$ et $R_{mur} = -10$. Ici au contraire, il est préférable d'augmenter la distance parcourue, mais de rencontrer un mur plutôt qu'un bord.



(c) Labyrinthe avec $R_{bord} = -25$ et $R_{mur} = -25$. Ici les deux obstacles sont tout aussi pénalisant, mais on choisit donc de minimiser les chocs même si cela demande de rallonger un peu la distance.

FIG. 4.9 – Labyrinthe avec obstacles et vent

NB : Les collisions sont signalées sur les graphiques par un cercle vert.

4.5 Interface graphique du Windy Grid World

Nous pouvons regarder comment l'interface graphique est implémenter sur le cas particulier de la Windy Grid World.

TD_Interface.sce Cet exécutable va lancer le modèle demandé par l'utilisateur. Pour cela il se base sur la variable *_model* gérée par RLtoolbox.tcl, son numéro correspond au numéros de la variable *Items* définit dans *TD_Interface.sce*.

MDL_edit.sce Cet exécutable Scilab gère l'affichage des éléments de la fenêtre de l'éditeur d'environnement ansi que leur comportement en effectuant les appels aux fonctions adéquates se trouvant dans le fichier *MdL_edit_functions.sci*.

Display_res.sce Cet exécutable Scilab gère l'affichage du résultat : environnement plus chemin choisit par l'agent. Il utilise les fonctions se trouvant dans le fichier *MdL_edit_functions.sci*.

MdL_edit_functions.sci Contient des fonctions appelées par l'éditeur d'environnement. Ces fonctions sont :

Pos_center=Center_calc(State) Calcul les coordonnées du centre de la case représentant la case *State*. *Pos_center* est une matrice de taille 2,1 avec l'abscisse x comme premier élément et l'ordonnée y comme deuxième.

draw_term(x0,y0,x1,y1) Dessine un état terminal (disque jaune). *(x0,y0)* représente le coin supérieur gauche du rectangle contenant l'état terminal et *(x1,y1)* le coin inférieur droit. Attention x0,y0,x1,y1 doivent être sous forme de chaines de caractères.

draw_wall(x0,y0,x1,y1) Dessine un mur (fond rouge avec bandes blanches). *(x0,y0)* représente le coin supérieur gauche du mur et *(x1,y1)* le coin inférieur droit. Attention x0,y0,x1,y1 doivent être sous forme de chaines de caractères.

draw_cliff(x0,y0,x1,y1) Dessine un cliff (case noire). *(x0,y0)* représente le coin supérieur gauche du cliff et *(x1,y1)* le coin inférieur droit. Attention x0,y0,x1,y1 doivent être sous forme de chaines de caractères.

draw_hole(x0,y0,x1,y1) Dessine un trou (fond noir avec disque blanc au centre). *(x0,y0)* représente le coin supérieur gauche du trou et *(x1,y1)* le coin inférieur droit. Attention x0,y0,x1,y1 doivent être sous forme de chaines de caractères.

draw_wind() Dessine les flèches représentant le vent. Cette fonction se base sur les matrices (chargées en mémoire lors du chargement du modèle) appelées *wind_X* et *wind_Y*.

a=mat2string(Mat) convertit une matrice en une chaine de caractère destinée à être écrite sur plusieurs lignes dans un fichier par une commande *tcl*.

Save() fonction permettant de sauver le modèle en cours d'édition dans un fichier. Cette fonction est appelé par la fonction APPLY en mode 2 (c'est à dire sauvegarde). Cela permet de s'assurer de tout mettre à jour avant de sauver.

Apply(MODE) Met à jour tous les champs et variables gérés par l'éditeur (vent , rewards, ...). *MODE* est un entier permettant de savoir dans quel contexte on souhaite faire la mise à jour. Si *MODE==1* on effectue juste une mise à jour de l'affichage. Si *MODE==2* on met a jour puis on lance la fonction de sauvegarde et si *MODE==3* on lance une simulation de l'environnement après la mise à jour.

Simul() Lance la simulation de l'environnement en cours d'édition. Cette fonction est appelé par la fonction APPLY en mode 3. Cela permet de s'assurer de tout mettre a jour avant de simuler.

First_State=SetStart(x,y) Cette fonction retourne dans la variable *First_State* la position du départ calculée à partir des coordonnées x et y qui sont passées en paramètres.

Export() Permet d'exporter au format PostScript le contenu de la fenêtre de dessin (canevas).

[TS,O,FS]=DRAW_ELTS(x,y) Fonction qui gère l'affichage ou l'effacement d'un nouvel élément en fonction du radiobutton choisi (start, finish, wall, cliff et hole). Cette fonction utilise *SetElts* en lui passant les coordonnées et le type d'élement en cours et retourne *erm_States* dans *TS*, *Obstacles* dans *O* et *First_State* dans *FS*.

Retour=SetElts(x,y,Elts_States,Elts) Dessine ou efface (suivant que l'élément existe ou pas dans la case cliquée) l'élément spécifié par *Elts* (état de fin, mur, cliff, ...). Cette fonction met aussi à jour la matrice correspondant aux placement des éléments (Term_States ou Obstacles) et la retourne.

DRAW_TERM(Term_States) Effectue l'affichage de tous les états terminaux en se basant sur la matrice *Term_States*.

DRAW_WALL(Obstacles) Effectue l'affichage de tous les *walls* en se basant sur la matrice *Obstacles*.

DRAW_CLIFF(Obstacles) Effectue l'affichage de tous les *cliffs* en se basant sur la matrice *Obstacles*.

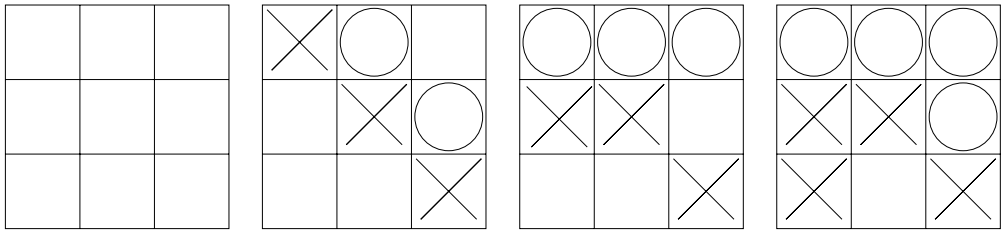
DRAW_HOLE(Obstacles) Effectue l'affichage de tous les *holes* en se basant sur la matrice *Obstacles*.

Chapitre 5

Jeu du Tic Tac Toe

5.1 Description du jeu

Le jeu du Tic Tac Toe[1] se joue habituellement sur un tableau 3x3 à deux joueurs. Cependant, il est possible de jouer sur des grilles plus grandes (pas forcément carrées), mais cela ne change rien à la méthode de résolution du problème. Chaque joueur joue alternativement et pose une "marque" sur une case libre. Généralement, une marque est une croix pour un joueur et un rond pour l'autre. La partie se termine quand un joueur a réussi à aligner 3 marques (ou plus sur une grille plus grande) horizontalement, verticalement ou diagonalement. Le jeu peut aussi se finir avec un "Draw", c'est-à-dire quand aucun joueur n'a gagné et que la grille est pleine. Nous pouvons voir sur la figure 5.1 plusieurs cas possibles.



(a) Une grille vide. (b) Le joueur avec les croix a gagné. (c) Le joueur avec les ronds a gagné. (d) Aucun joueur n'a pu gagner. C'est un Draw.

FIG. 5.1 – Exemples de jeux du Tic-Tac-Toe

5.2 Résolution par RL

5.2.1 Description de la méthode

Nous allons utiliser une méthode d'apprentissage par renforcement de manière à avoir un joueur intelligent capable d'éviter des pièges. Pour cela nous devons définir une fonction ,V, qui pour chaque état du jeu associera une valeur indiquant la désirabilité de l'état. Un état est bien entendu une configuration possible de la grille, et V peut être la probabilité de gagner à partir de cet état.

Evidemment, nous connaissons à priori cette probabilité que pour les états terminaux, où l'on a $V(s) = 1$ si l'état est gagnant, et $V(s) = 0$ si l'état est perdant. En cas de Draw, on peut

[1] Plus communément appelé Morpion

considérer que l'on a évité le pire, et on peut mettre $V(s) = 1$. Cela va nous permettre d'avoir un agent qui cherchera à ne jamais perdre, mais pas forcément à gagner. Après chaque partie, on peut donc modifier les probabilités de gagner pour chacun des états intermédiaires avec la formule :

$$V(s) = V(s) + \alpha[V(s') - V(s)] \tag{5.1}$$

avec s' l'état courant, s l'état précédent et α une petite fraction positive.

Nous choisissons d'initialiser V à 0.5 pour tous les états non terminaux.

On peut noter que pour un joueur donné, seul l'état après son déplacement importe, on parle donc d'*afterstate*. L'agent doit donc déterminer à chaque étape quel mouvement lui permettra d'atteindre l'afterstate avec la plus grande valeur. Cependant ce choix glouton est insuffisant en phase d'apprentissage, car en procédant ainsi, nous n'explorerons pas tous les états. Nous devons donc introduire des mouvements exploratoires pour la phase d'apprentissage. Cette méthode est celle employée par Sebastian Siegel lors de son projet sur le Tic-Tac-Toe[2].

5.2.2 Compression des états

Le plus simple pour gérer V est de maintenir une table de correspondance (lookuptable) entre les états et leur valeur de V. Cependant, ceci pose un problème, la taille de cette lookuptable. Pour une grille 3x3 nous avons déjà 5890 états, et si on augmente la taille de la grille, ce nombre d'état augmente de manière drastique. Procéder ainsi va donc demander beaucoup de mémoire et aussi beaucoup de temps CPU.

Il est possible de réduire ce nombre d'états en utilisant les symétries de la grille. Comme nous pouvons le voir sur la figure 5.2 il peut y avoir jusqu'à 8 états qui sont équivalents du point de vue leur valeur V et l'on peut passer de l'un à l'autre avec seulement une rotation et/ou une symétrie horizontale ou verticale. Notre objectif est donc de n'avoir qu'une seule entrée dans la lookuptable

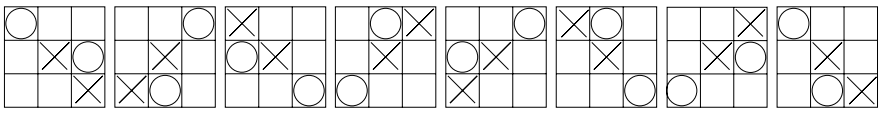


FIG. 5.2 – Exemple d'états équivalents par symétrie et rotation

pour ces huits états. Pour cela nous introduisons deux fonctions : $S(X)$ et $S(O)$ respectivement pour les joueurs avec les croix et ceux avec les ronds. Pour calculer ces fonctions, nous avons besoin de numéroter chaque case de la grille comme indiqué sur la figure 5.3. Au final nous avons $S(X) = \sum_{i=1}^{9} 0.5^i . I_X(i)$ pour une grille 3x3 et avec $I_X(i) = 1$ si il y a un X sur la case i et $I_X(i) = 0$ sinon. De manière similaire, $S(O) = \sum_{i=1}^{9} 0.5^i . I_O(i)$.

Nous choisissons comme représentant de ces huit états celui avec le plus petit $S(X)$ et au cas où plusieur états auraient le même $S(X)$ on choisira parmis eux celui avec le plus petit $S(O)$. Ainsi nous pouvons réduire le nombre d'état à 825 (au lieu de 5890 au départ)[3].

5.3 Interprétation des résultats

Deux types de résultats sont disponibles actuellement :

[2] Voir *Training an artificial neural network to play tic tac toe*, Sebastian Siegel, ECE 539 Term Project - `http ://www.cae.wisc.edu/ ece539/project/f01/`

[3] Plus de détails sont disponible dans le rapport de projet *Training an artificial neural network to play tic tac toe*, Sebastian Siegel, ECE 539 Term Project - `http ://www.cae.wisc.edu/ ece539/project/f01/`

1	2	3
4	5	6
7	8	9

FIG. 5.3 – Numérotation de la grille

1. Les résultats en fin d'entrainement qui nous permettent de savoir comment s'est passé l'entrainement, et en particulier le nombre d'états que l'on a pu explorer.
2. Les résultats issus d'un affrontement entre deux ordinateurs que l'on a entrainé de manière indépendante. En fait, on donne une lookuptable différente à deux joueurs, ils s'affrontent pendant un certain nombre de parties, et on regarde les victoires de chacun ainsi que les draw.

Aprés un entrainement de 3000 jeux nous avons les résultats présents sur le graphique 5.4. Nous pouvons constater qu'au début de l'apprentissage les joueurs arrivent à se gagner l'un l'autre et ne font presque pas de draw. Cela vient du fait qu'ils n'ont pas encore exploré beaucoup d'états. Au contraire, au fur et à mesure des parties, les joueurs commencent à connaitre plus de statégies, grâce aux mouvements aléatoires, et deviennent plus apte à contrer les attaques de leur adversaire. Dans ce cas de figure, il est normal de terminer le jeu avec un draw.

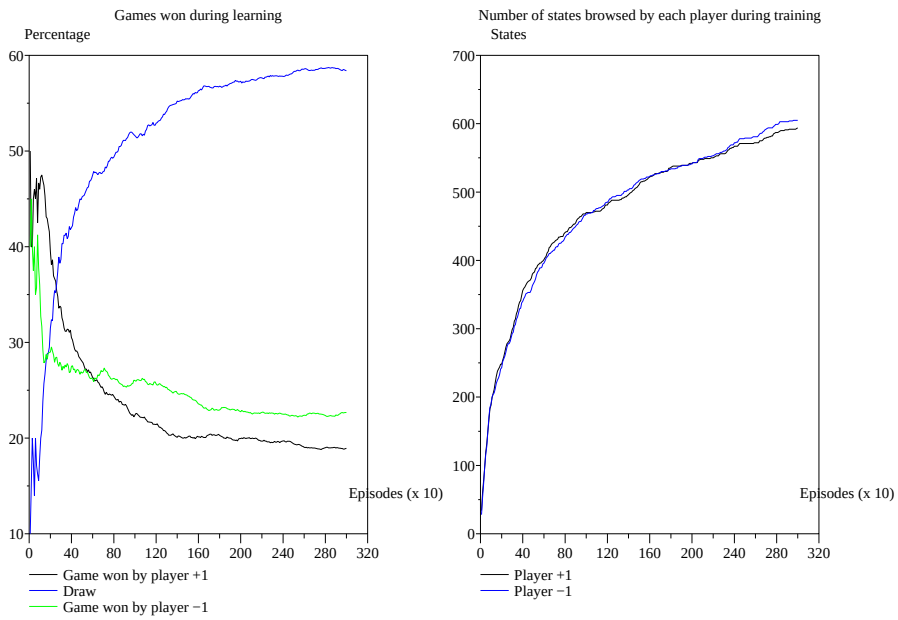


FIG. 5.4 – Résultat après un entrainement de 3000 parties

Au bout des 3000 jeux, on a donc 60% de draw, et on a exploré environ 650 états sur les 825 possibles. Cependant, si on poursuit l'entrainement (environ 10000 parties), on arrive a explorer

près de 800 états, mais la répartition entre les draw et les victoires de chaques joueurs reste identique.

Tout d'abord, nous avons fait s'affronter deux joueurs avec le même niveau de connaissance du jeu, i.e. les mêmes lookuptable. Comme nous pouvions nous y attendre les parties se terminent à 100% par des draw, surtout quand les joueurs ont été entrainés pendant longtemps.

Deux types de joueurs peuvent être modélisés seulement en changeant la récompense attribuée à un draw. Jusqu'ici nous avons utilisé une récompense de 1 pour ce cas, ce qui fait que nous avons un joueur plutôt défensif. En effet, il essait de ne jamais perdre, mais si il se retrouve dans une situation où il a le choix entre gagner ou faire un draw, il choisira l'un ou l'autre indifféremment. En revanche, si on accorde une récompense de 0 pour un draw, on a un joueur plutôt attaquant étant donné que la seule situation qui l'intéresse est la victoire.

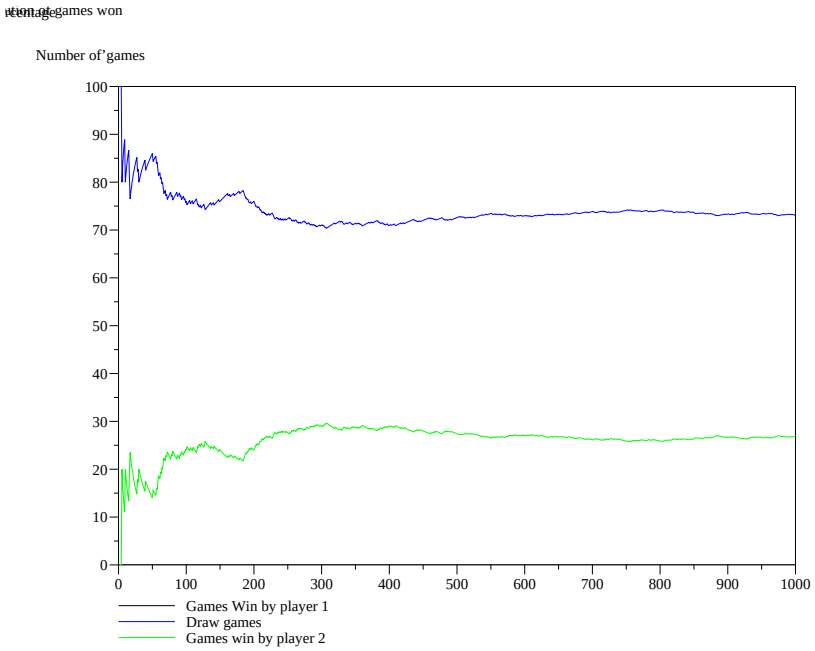


FIG. 5.5 – Benchmark entre un joueur défensif (joueur 2) et un offensif (joueur 1) ayant chacun appris sur 10000 parties

En pratique, nous faisons affronter 2 joueurs qui ont appris sur 10000 parties, mais l'un est défensif et l'autre offensif. Nous pouvons voir sur la figure 5.5 que le joueur 2 qui est défensif est le seul à gagner des parties, à hauteur de 30%, et que 70% des parties se concluent sur un draw. Ceci s'explique par le fait que je joueur 1 essaie seulement de gagner, mais il n'arrive pas a vaincre la defense du joueur 2, qui est parfaite à ce niveau d'entrainement.

La "qualité" d'un joueur dépend intuitivement du nombre de parties d'apprentissage qu'il aura effectuer. C'est ce que nous vérifions sur le graphique 5.6. En effet, ici nous avons deux joueurs défensifs qui s'affrontent, mais le joueur 1 a bénéficié d'un apprentissage de 10000 parties, tandis que le joueur 2 n'a eu que 100 parties. Les résultats obtenus confirment ce que nous pensions, le joueur 2 n'arrive pas à gagner, mais seulement à faire des draw (65%). Néanmoins, il faut nuancer ce résultat, car le joueur 1 étant défensif, il considère qu'un draw est aussi bien qu'une victoire et donc il lui arrive très certainement de choisir de faire un draw, alors qu'il aurait pu facilement

gagner.

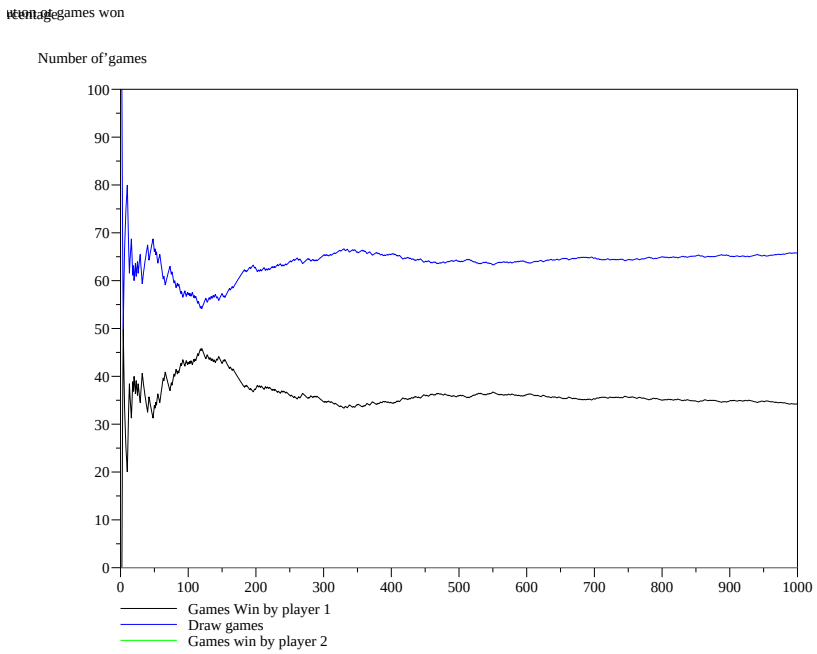


FIG. 5.6 – Benchmark entre un joueur expert (10000 parties d'apprentissage, joueur 1) et un joueur débutant (100 partie d'apprentissage, joueur 2). Les deux joueurs sont du type défensif.

Nous pouvons vérifier cette hypothèse en utilisant un joueur 1 attaquant. Sur la figure 5.7, nous voyons que dans ce cas, le joueur 1 a amélioré sa probabilité de gagner, environ 50%. En revanche, il prend tellement de risques que le joueur 2 arrive parfois à gagner, ce qui n'était pas le cas précedemment.

Un joueur avec une récompense de 0.5 pour un draw paraît donc être le meilleur compromis pour obtenir un joueur qui sait à la fois se défendre pour éviter de perdre et gagner quand il en a l'opportunité. Après un apprentissage sur 4000 parties nous avons donc effectué des tests avec les autres agents pour voir les résultats obtenus et verifier qu'ils concordent avec ceux attendus. Les courbes obtenues se trouvent en 5.8 (page 47).

On peut donc remarquer sur les courbes de la figure 5.8 (page 47) que les résultats obtenus sont ceux attendus. En effet, le joueur ayant eu une récompense de 0.5 pour un draw arrive à gagner les deux autres types d'agent.

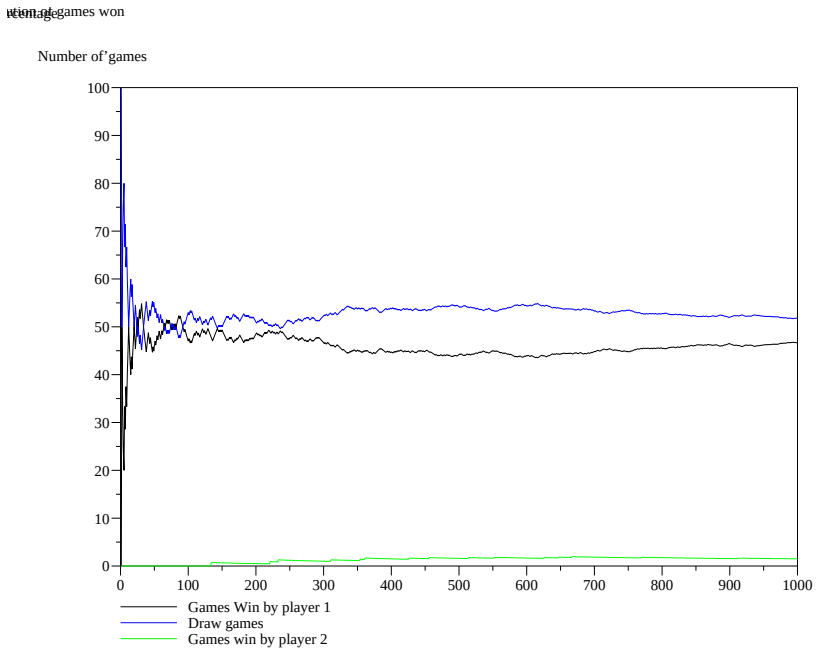
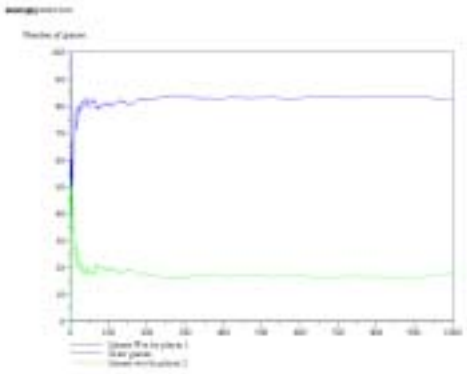
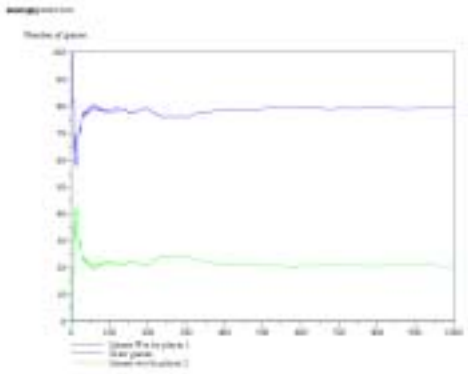


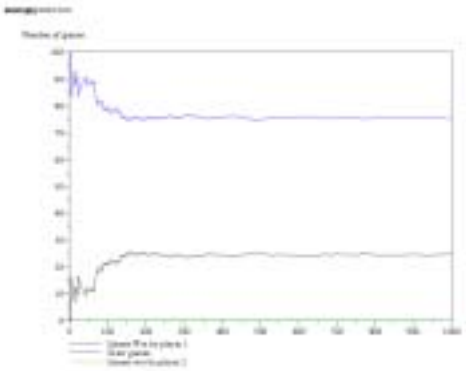
FIG. 5.7 – Benchmark entre un joueur expert offensif (10000 parties d'apprentissage et récompense de draw à 0, joueur 1) et un joueur débutant défensif(100 parties d'apprentissage et une récompense de draw à 1, joueur 2).



(a) Benchmark entre un joueur défensif (joueur 2) et un offensif (joueur 1)



(b) Benchmark entre un joueur mi-offensif mi-défensif (joueur 2) et un offensif (joueur 1)



(c) Benchmark entre un joueur mi-offensif mi-défensif (joueur 1) et un défensif (joueur 2)

FIG. 5.8 – Comparaison entre différents agents ayant appris sur 4000 parties

Liste des illustrations

Liste des algorithmes