1.5em 0pt

Apprentissage par Renforcement

Lounès Meddahi

10 juin 2021

Table des matières

Table des figures

1 Introduction

L'objectif de ce compte rendu est de présenter certains algorithmes d'apprentissage par renforcement, leurs implémentations ainsi qu'une étude de leurs compléxités. Une implémentation réalisée en python peut-être retrouvée sur ce dépôt Github.

Pour les premiers algorithmes, on a utilisé une table de hachage afin de représenter les fonctions de transitions et de retour. Ces algorithmes peuvent être adaptés afin d'utiliser des listes plutôt que des tables de hachages.

2 Notations

— un ensemble d'états du jeu, s ∈ S

— un ensemble d'actions possibles, a ∈ A

— P est la fonction de transition : *pour chaque couple (état, action), cette fonction indique la probabilité que le système soit ensuite dans chaque état* :

$$P : S \times A \times S \to [0, 1]$$

— R est la fonction de retour : *à valeur réelle, cette fonction formalise les conséquences d'une action émise dans un état.* :

$$R : S \times A \times S \to \mathbb{R}$$

— une politique déterministe : *à un état, la politique associe une action :*

$$\pi : S \to A$$

— une politique stochastique : *à un état, la politique associe une distribution de probabilités sur les actions :*

$$\pi : S \times A \to [0, 1]$$

— V^{π}(s) est la valeur pour chaque état s d'un environnement pour une politique π : *c'est l'espérance de R quand l'environnement est initialement dans l'état s :*

$$V^{\pi(s)}$$

= E[R s_0= s]

— Q^{π}(s,a) est la qualité d'une paire état action action (s,a) pour une politique π : *ça nous permet d'étudier l'action a une fois faite dans l'état s :*

$$Q^{\pi}$$

(s,a) = E[V^{π}(s_0) | $s_0 = s, a_0 = a, a_{t>0} \sim \pi$]

3 Algorithmes

3.1 Algorithme n°0 : calcul de la valeur

3.1.1 Idée générale

Ce premier algorithme a pour but de calculer la valeur d'une politique grâce à l'équation de Bellman d'écrite ci-dessous :

$$V^{\pi}(s) = \sum_{a \in A} \pi(s,a) \sum_{s' \in S} P(s,a,s')[R(s,a,s') + \gamma V^{\pi}(s')] \tag{1}$$

Un algorithme simple pour faire cette tâche peut être :

1. Initialiser : V(s) $\leftarrow$ 0 , $\forall$ s $\in$ S , k $\leftarrow$ 0
2. Itérer : $V_{k+1}(s) = \sum_{a \in A} \pi$(s,a) $\sum_{s' \in S}$P(s,a,s')[R(s,a,s') + γ $V^{\pi}(s')$] ; k+1
3. Jusqu'à ce que l'écart entre V_k et V_{k-1} soit faible

3.1.2 Implémentation

On peut retrouver cet algorithme dans le fichier *Algo0.py*.
Dans ce fichier on y retrouve les fonctions :

— *politique_taxi*() qui produit la politique stochastique du taxi (sous forme d'un dictionnaire (Etat,{action :probabilité}))
— *transition_taxi*() qui produit la fonction de transition du taxi (sous forme d'un dictionnaire (Etat,(action,{Etat :probabilité}))
— *retour_taxi*() qui produit la fonction de retour du taxi (sous forme d'un dictionnaire (Etat,(action,{Etat :retour}))
— *valeur*(*politique*, *actions*, *etats*, *transitions*, *retour*, *gamma*) qui permet d'avoir la valeur associée à la politique passée en paramètre en utilisant l'environnement décrit en paramètre (actions , etats , transitions , retour), où gamma est le facteur déprécié

3.1.3 Résultats

A la fin du programme, un code a été ajouté afin de trouver les valeurs données dans le document source :
$'A'$: 87.43423712514684,$'B'$: 98.56278129353218,$'C'$: 87.39517337027189

On a donc bien trouvé que son revenu à long terme sera en moyenne de 87,43 s'il démarre dans l'état A, 98,6 s'il démarre dans l'état B et 87,40 s'il démarre dans l'état C.

Un représentation de la fonction valeur de toutes les politiques stationnaires et déterministes possibles du problème du chauffeur de taxi, pour γ = 0,9 est donnée ci-dessous.

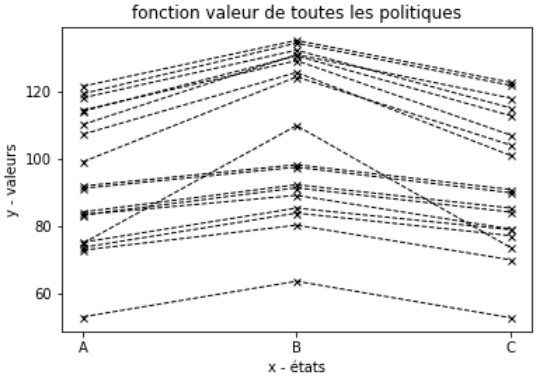


FIGURE 1 – fonction valeur de toutes les politiques

3.2 Algorithme n°1 : itération sur les politiques

3.2.1 Idée générale

Cet algorithme a pour but de calculer la valeur ϵ-optimale d'un environnement grâce à des politiques aléatoires et à la formule :

$$\sum_{s' \in S} P(s, a, s')[R(s, a, s') + \gamma V^{\pi}(s')] \tag{2}$$

Pour ce faire, on générera des politiques aléatoires et garderons la politique qui nous donne la meilleur valeur, et ça, tant que l'écart entre V^{π} et V^{*} ne nous convient pas.

On trouvera ainsi une politique π telle que $\| V^{\pi}\text{-}V^{*} \|_{\infty} \leq \epsilon$.

3.2.2 Algorithme

Algorithm 1 L'algorithme d'itération sur les politiques.

Require: un PDM : $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$
Require: un seuil de précision ϵ
initialiser π_0 (aléatoirement ou autrement)
$k \leftarrow 0$
repeat
  initialiser V_0^{π} (aléatoirement ou autrement)
  $i \leftarrow 0$
  repeat
    for tout état $s \in \mathcal{S}$ **do**
      $V_{i+1}^{\pi_k}(s) \leftarrow \sum_{s' \in \mathcal{S}} \mathcal{P}(s, \pi_k(s), s')[\mathcal{R}(s, \pi(s), s') + \gamma V_i^{\pi_k}(s')]$
    end for
    $i \leftarrow i + 1$
  until $||V_i^{\pi_k} - V_{i-1}^{\pi_k}||_{\infty} \leq \epsilon \frac{(1-\gamma)}{2\gamma}$
  for tout état $s \in \mathcal{S}$ **do**
    $\pi_{k+1}(s) \leftarrow \arg\max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}(s, a, s')[\mathcal{R}(s, a, s') + \gamma V^{\pi_k}(s')]$
  end for
  $k \leftarrow k + 1$
until $\pi_k = \pi_{k-1}$

FIGURE 2 – Algorithme 1

3.2.3 Implémentation

Cet algorithme est disponible dans le fichier *Algo*1.*py*. Ce fichier réutilise les fonctions transition_taxi() et retour_taxi() du fichier *Algo*0.*py*. Dans ce fichier, on y retrouve également les fonctions :

- *random_pol*(*actions*) qui renvoie une action aléatoires parmi les actions possibles
- *IterationPolitiques*(*etats*, *actions*, *transitions*, *retour*, *gamma*, *epsilon*) qui renvoie la politique ϵ-optimale par la méthode par itération sur les politiques

3.2.4 Résultats

A la fin du programme, un code a été ajouté afin de trouver une politique 0.01-optimale.

On trouve comme politique (déterministe) :

'A' : 'a2', 'B' : 'a3', 'C' : 'a2'

Et les valeurs qu'on trouve pour cet politique sont :

'A' : 121.64862519185584, 'B' : 135.30142959222255, 'C' : 122.83205714451859

(Qui sont meilleurs que celles trouvées précédemment).

Cette politique détérministe peut-être vue comme une politique stochastique avec la répartition de probabilité suivante :

- politique['A'] = 'a1' :0 , 'a2' :1 , 'a3' :0
- politique['B'] = 'a1' :0 , 'a3' :1
- politique['C'] = 'a1' :0 , 'a2' :1 , 'a3' :0

3.3 Algorithme n°2 : itération sur la valeur

3.3.1 Idée générale :

Cet algorithme a pour but de calculer la valeur ϵ-optimale d'un environnement grâce à l'équation d'optimalité de Bellman :

$$\max_{a \in A(s)} \sum_{s' \in S} P(s,a,s')[R(s,a,s') + \gamma V^{\pi}(s')] \tag{3}$$

3.3.2 Algorithme

Algorithm 2 L'algorithme d'itération sur la valeur.

Require: un PDM : $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$
Require: un seuil de précision ϵ
initialiser $V_0 \leftarrow 0$
$k \leftarrow 0$
repeat
  for tout état $s \in \mathcal{S}$ **do**
    $V_{k+1}(s) \leftarrow \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}(s,a,s')[\mathcal{R}(s,a,s') + \gamma V_k(s')]$
  end for
  $k \leftarrow k + 1$
until $||V_k - V_{k-1}||_{\infty} \leq \frac{\epsilon(1-\gamma)}{2\gamma}$
for tout état $s \in \mathcal{S}$ **do**
  $\pi(s) \leftarrow \arg\max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}(s,a,s')[\mathcal{R}(s,a,s') + \gamma V(s')]$
end for

FIGURE 3 – Algorithme 2

3.3.3 Implémentation

Cet algorithme est disponible dans le fichier Algo2.py. Ce fichier réutilise les fonctions transition_taxi() et retour_taxi() du fichier Algo0.py.

— IterationValeur(etats , actions , transitions , retour , gamma , epsilon) qui renvoie la politique ϵ-optimale par la méthode par itération sur les politiques

3.3.4 Résultats

A la fin du programme, un code a été ajouté afin de trouver une politique 0.01-optimale.
On trouve comme politique (déterministe) :
'A' : 'a2', 'B' : 'a3', 'C' : 'a2'
Et les valeurs qu'on trouve pour cet politique sont :
'A' : 121.64862519185584, 'B' : 135.30142959222255, 'C' : 122.83205714451859
(Comme à la partie 2.2.4).

3.4 Algorithme n°3 : algorithme TD pour une politique déterministe

3.4.1 Idée générale :

Cet algorithme a pour but de s'approcher de la valeur optimale sans connaitre les fonctions de transitions et de retours. Pour ce faire, on va faire un enchaînement d'actions en connaissant la politique et observer le retour ainsi que l'état d'arrivé.

Pour ce faire, nous allons utiliser l'équation :

$$V^{\pi}(s_t) = V^{\pi}(s_t) + \alpha(s_t, n(s_t))[r_t + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t)] \tag{4}$$

tel que :

$$\alpha(s, n(s)) = \frac{1}{1 + n(s)} \tag{5}$$

Pour cett partie, on va utiliser pour la 1ère fois des listes, ce qui va nous donner une implémentation "moins" naturelle qu'avant car à partir de maintenant on va attribuer à chaque état et à chaque action une valeur entière (et positive si possible) afin de parcourir une liste.

On va pouvoir transformer cet algorithme en algorithme pour une politique stochiastique si besoin.

3.4.2 Algorithme

Algorithm 3 L'algorithme TD pour une politique déterministe (et stationnaire).

Require: $\mathcal{S}$, $\mathcal{A}$, γ et une politique π.

$\widehat{V}^{\pi}(s) \leftarrow 0, \forall s \in \mathcal{S}$
$n(s) \leftarrow 0, \forall s \in \mathcal{S}$
repeat
  initialiser l'état initial s_0
  $t \leftarrow 0$
  repeat
    émettre l'action $a_t = \pi(s_t)$
    observer r_t et s_{t+1}
    $\widehat{V}^{\pi}(s_t) \leftarrow \widehat{V}^{\pi}(s) + \alpha(s_t, n(s_t))[r_t + \gamma \widehat{V}^{\pi}(s_{t+1}) - \widehat{V}^{\pi}(s_t)]$
    $n(s_t) \leftarrow n(s_t) + 1$
    $t \leftarrow t + 1$
  until s_t est un état final
until critère d'arrêt vérifié.

FIGURE 4 – Algorithme 3

3.4.3 Implémentation

Cet algorithme est disponible dans le fichier Algo3.py. Pour cette implémentation on va utiliser des listes. Donc chaque état ainsi que chaque action est modélisée par un nombre entier dans l'exemple du taxi. Ce fichier utilise les fonctions :

- etats() afin d'avoir la liste des états possibles
- etat_suivant(etat , action) qui renvoie la liste des états suivants possibles avec leur probabilité et leur retour (après avoir fait l'action action dans l'état etat)
- politique_taxi() qui renvoie la politique déterministe de l'environnment
- actions() afin d'avoir la liste des actions possibles
- init() est la fonction qui donne l'état inital
- td(etats , actions , gamma , politique , init , est_final , etat_suivant) est l'implémentation de l'algorithme. Cette fonction utilise une liste d'états, d'actions, un facteur déprécié, une politique, une fonction init qui permet d'initialiser l'état s_0, une fonction est_final() afin de savoir si on est dans un état final ou non, etat_suivant() qui est une fonction qui permet d'avoir, pour une action effectuée dans un état s_t à l'instant t, le retour r_t ainsi que l'état s_{t+1}

3.4.4 Résultats

——————Informations : ——————

La seule chose qui peut poser problème si on veut utiliser cet algorithme pour un autre problème est la dimension du problème.

Par exemple ici on est en dimension 1, car un état correspond à un entier. Alors qu'on aurait pu avoir des coordonnées comme état (ex : Labyrinthe). Une modélisation afin de généraliser ça est à venir pour cet algorithme, mais l'idée est qu'il faut utiliser une fonction valeur à même dimension que l'environnement, car comme ça, peut import la dimension de l'environnement, on va pouvoir avoir une représentaion dans un repère de même dimension.

——————Problèmes : ——————

Cet algorithme pose problème car ici après énormément d'itération en utilisant la même politique que dans les fonctions précédentes (peut-être faut-il attendre beaucoup plus pour avoir la valeur attendue).

A la fin du fichier, un programme est utilisé afin d'essayer cet algorithme pour l'exemple du chauffeur de taxi avec la même politique détérministe que les algorithmes précédants (la politique qui va nous permettre d'avoir les meilleurs résultats).

3.5 Algorithme n°4 : algorithme Q-Learning

3.5.1 Idée générale :

L'algorithme Q-Learning est le premier des algorithmes d'apprentissage implémenté jusqu'à maintenant. Ce qui signifie que désormais on ne va plus connaitre la politique, la fonction de transition et de retour, car on va observer l'évolution de l'agent dans l'environnement.
Pour ce faire, l'algorithme va effectuer des "épisodes", qui correspondent à un passage dans l'environnement en partant d'un état initial pour arriver à un état final en faisant un enchainement d'actions. Le but étant de minimiser le nombre actions à faire à faire de maximiser ou minimiser l'objectif.

C'est donc à partir de maintenant que les premiers problèmes de l'apprentissage par renforcemment commencent à arriver. Comment choisir l'action à faire ; Comment savoir si on doit commencer la phase d'exploitation ; Comment savoir si on doit continuer la phase d'exploration.

3.5.2 Algorithme

Algorithm 4 L'algorithme *Q-Learning*.

Require: $\mathcal{S}$, $\mathcal{A}$, γ.

$\widehat{Q}(s,a) \leftarrow 0, \forall (s,a)$
$n(s,a) \leftarrow 0, \forall (s,a) \in (\mathcal{S}, \mathcal{A})$
repeat
  initialiser l'état initial s_0
  $t \leftarrow 0$
  while épisode non terminé **do**
    sélectionner l'action a_t et l'émettre
    observer r_t et s_{t+1}
    $\widehat{Q}(s_t,a_t) \leftarrow \widehat{Q}(s_t,a_t) + \alpha(s_t,a_t,n(s_t,a_t))[r_t + \gamma \max_{a' \in \mathcal{A}} \widehat{Q}(s_{t+1},a') - \widehat{Q}(s_t,a_t)]$
    $n(s_t,a_t) \leftarrow n(s_t,a_t) + 1$
    $t \leftarrow t + 1$
  end while
until critère d'arrêt vérifié.

FIGURE 5 – Algorithme 4

3.5.3 Implémentation

Comme dit dans l'introduction, cet algorithme va chercher à minimiser le nombre d'actions à faire pour arriver à l'état final. Le problème dans cette partie est donc : Comment choisir ces actions ?

Pour ce faire, il existe plusieur façon. Voici quelques possibilités :

— **aléatoire** : Cette méthode consiste à choisir une action aléatoire lorsque l'agent arrive dans un nouvel état. Le problème de ce choix est donc le fait qu'au premier épisode ou au dernier, l'agent ne va jamais utiliser ce qu'il a déjà fait lors des épisodes précédents.

— **gloutonne** : L'agent va toujours choisir l'action qui permet d'avoir le meilleur retour à partir de son état actuel, soit $a_{gloutonne}(s_t) = arg\max_{a \in A(s_t)} Q(s_t,a)$. Le problème ici étant le fait que la phase d'exploration sera presque nulle lorsqu'une première solution sera trouvée pour arriver à l'objectif.

— **Boltzmann** : Cette manière de choisir est un cas particulier de la séléction $softman$ qui permet de donner une certaine distribution de probabilité aux différentes actions en fonction de leur qualité. La répartition de Boltzmann est donnée par l'équation :

$$Pr[a_t \mid s_t] = \frac{e^{\frac{Q(s_t,a_t)}{\tau}}}{\sum_{a \in A(s_t)} e^{\frac{Q(s_t,a_t)}{\tau}}}$$

Avec τ un réel positif appelé *température*. Plus cette valeur est grande, plus le choix des actions sera proche d'une séléction aléatoire. Mais plus cette valeur est faible, plus le choix des actions

sera similaire à un choix Glouton. Le but étant donc de trouver la bonne manière de choisir ce τ.

Cet algorithme est disponible dans le fichier *Algo4.py*.

Cette implémentation a encore été faite dans le but d'être utilisée avec des listes. Ainsi, chaque actions et états doivent être représenté par des valeurs entières. Cette fonction à été faite dans le but de faire tourner des envirnonnements de dimension 2 (ex : labyrinthe).

L'implémentation de la fonction de Q-Learning à pour signature :

$$Q\_Learning(etats, actions, gamma, execution, init, decision, final, environnement)$$

Afin de pouvoir y mettre l'ensemble des données qu'on veut pour définir notre environnement ainsi que la manière d'évoluer à l'intérieur (choix des actions, état initial, état d'arrivé).

Une autre fonction clé est la fonction *execution* qui permet d'éxecuter une certaine action a dans l'état s à l'instant t. Dans notre exemple de labyrinthe, on a 4 actions possibles :

- Aller à droite : Action 0
- Aller en haut : Action 1
- Aller à gauche : Action 2
- Aller en bas : Action 3

/ !\ Toutes ces actions ne sont pas forcément possible dans tous les états à cause des murs qui délimitent le labyrinthe.

3.5.4 Résultats

Dans ce fichier, on peut retrouver 2 programmes majeurs.

Le 1er permet d'utiliser le labyrinthe ci-dessous, pour faire 15 fois 70 épisodes en utilisant la stratégie de *Boltzmann* afin de produire un graphique qui montre l'évolution du nombre d'étape à faire pour arriver à l'état final du labyrinthe.

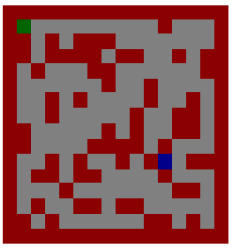
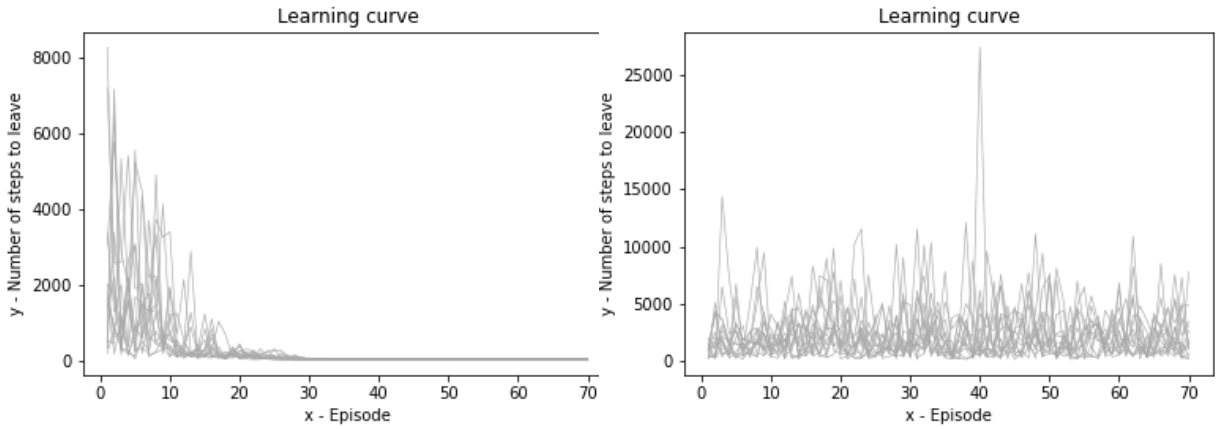


FIGURE 6 – Labyrinthe



(a) Fonction Boltzmann ($\tau = 1$)

(b) Fonction aléatoire

FIGURE 7 – Courbe d'apprentissage selon la méthode utilisée pour choisir l'action

Le second qui permet un affichage des mouvements de notre agent dans l'environnement toutes les 0.1 seconde afin de pouvoir constater sa rapidité à sortir du labyrinthe au fil des épisodes.

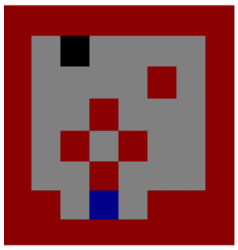
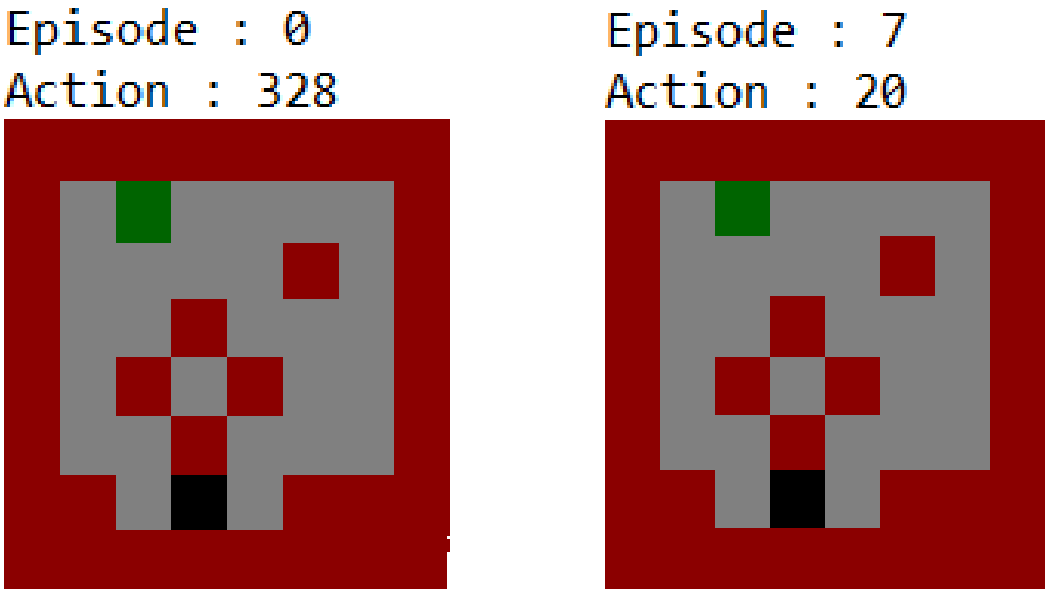


FIGURE 8 – Labyrinthe 2



(a) Episode 0 (b) Episode 7

FIGURE 9 – Affichage du labyrinthe au fil des épisodes

3.5.5 Réflections :

A venir.

(Partie qui va consister à étudier le choix de τ au fil des épisodes)
(Partie qui va étudier l'évolution de la courbe d'apprentissage en fonction des changements fait à l'environnement avec une fonction qualité déjà entrainée)
(Partie qui étudier un environnement $on - policy$ (prendre l'exemple du labyrinthe avec :
1)Une sortie qui se déplace selon un même patern
2)Une sortie qui se déplace selon un patern aléatoire
3)Une sortie qui se déplace en supprimant des murs (càd impact sur l'environnement)
)

4 Remarques

Pour les algorithmes 0,1 et 2, une table de hachage a été utilisée, mais cela peut poser problème pour des gros environnements dû au stockage des clés (actions et états). Ils seront donc adaptés prochainement avec des listes.

C'est pour cela qu'à partir de l'algorithme 3, des listes sont utilisées afin de ne plus avoir ce besoin de stocker les clés. Un autre point positif, c'est que désormais, on peut travailler plus facilement avec des environnement qui nécessite d'utiliser des coordonnées (ex : jeu du labyrinthe).

5 Compléxité

5.1 Compléxité théorique :

5.2 Compléxité Réel :

6 Bilan

A venir.