ULB UNIVERSITÉ LIBRE DE BRUXELLES

INFO-F311 - Projet d'IA 3

Apprentissage par renforcement

Auteur:
Manuel Rocca - 000596086

Professeurs:
Tom Lenaerts
Assistants:
Axel Abels
Martin Colot
Yannick Molinghen
Pascal Tribel

Année académique 2025-2026

Contents

1 Introduction

Pour ce troisième projet du cours d'Intelligence Artificielle, nous avons implémenté des algorithmes d'apprentissage par renforcement (*reinforcement learning* en anglais). Nous utilisons toujours l'environnement *LLE* comme pour les parties 1 et 2.

Trois différences (ou ajouts) sont introduites: les cases tourbillons (où l'agent meurt) la fonction de récompense, et surtout, l'aspect stochastique, non-déterministe de l'environnement.

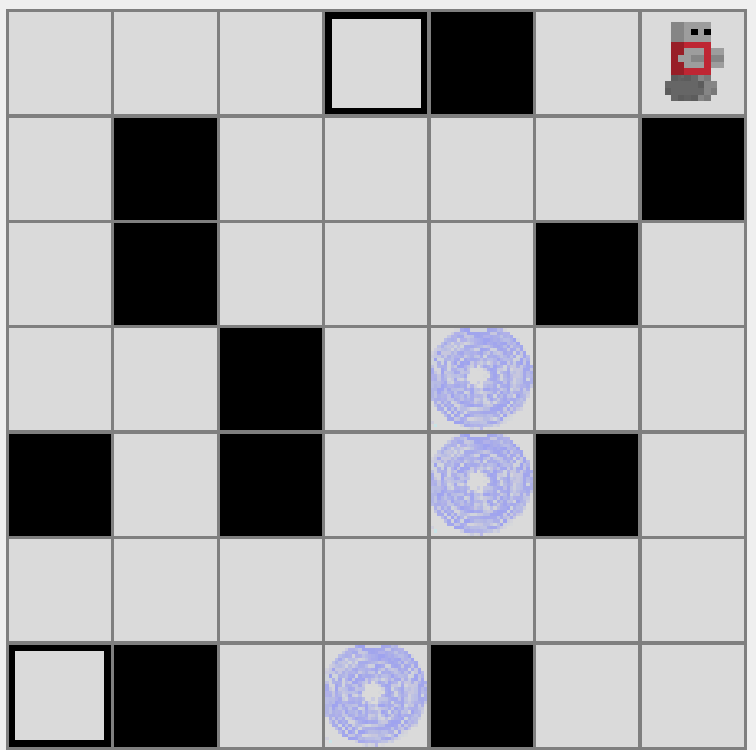


Figure 1: Environnement utilisé

1.1 Environnement stochastique

Un environnement stochastique est un environnement où, lorsque l'agent effectue une action, nous ne sommes pas certains de l'état dans lequel il va arriver. En pratique, si un agent dans un état s souhaite effectuer une action a pour arriver dans un nouvel état s', il y a une probabilité p qu'une autre action aléatoire soit prise.

1.2 La fonction de récompense

Une fonction de récompense est une fonction qui, pour chaque transition d'état, octroie une valeur. En d'autres termes, une valeur numérique est associée à chaque action prise par l'agent afin de le diriger de manière idéalement souhaitée.

Associer une valeur adéquate est tout sauf quelque chose de simple. Des comportements inattendu peuvent survenir si cette fonction est mal définie (par exemple: si un agent doit atteindre une sortie avec une récompense positive mais que le coût de vie, le coût par pas est très élevé, il aura plus tendance à chercher un moyen de faire le moins de pas possible en trouvant un endroit plus proche pour mourir permettant une minimisation de son score total).

2 Value Iteration

L'algorithme *Value Iteration* utilise l'équation de Bellman (1) de manière itérative pour mettre à jour les valeurs estimées de chaque état de l'environnement.

$$V^*(s) = \max_a \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V^*(s') \right] \tag{1}$$

Équation de Bellman caractérisant les valeurs optimales

Les itérations de l'algorithme sont bornées par une valeur δ comme suit:

$$\max_{s \in S} |V_{k+1}(s) - V_k(s)| < \delta \tag{2}$$

En d'autres termes, à chaque itération, la différence/variation entre les valeurs des états précédents et les nouvelles valeurs est calculée en appliquant l'équation 1. Si celle-ci est inférieure du seuil δ donné, l'algorithme s'arrête.
L'algorithme converge après un certain nombre d'itérations. Ce nombre peut être, dans certains cas, très grand voire quasi infini. C'est pourquoi il est intéressant d'établir un seuil de variation maximal afin d'arrêter l'algorithme après une durée d'exécution raisonnable. Comme nous l'avons vu au cours théorique, parfois un dixième des itérations suffisent à obtenir des valeurs pratiquement égales aux valeurs obtenues à la convergence.

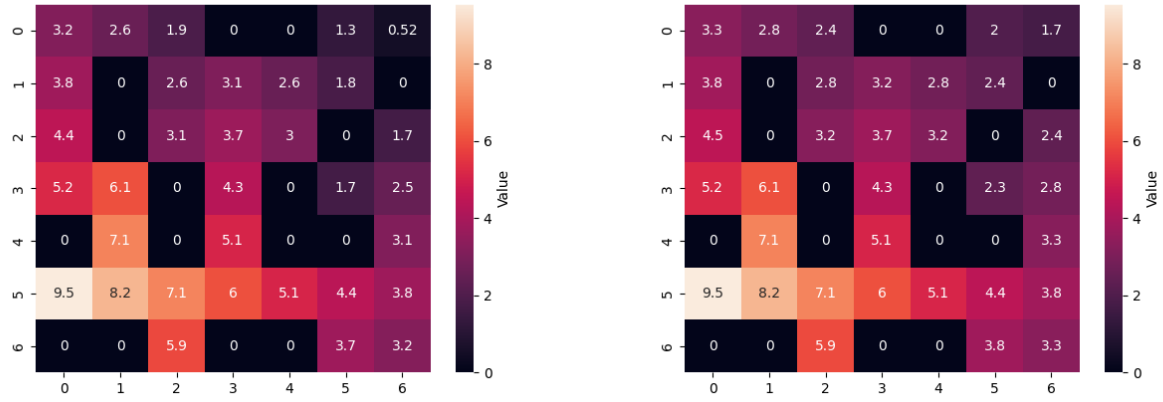
2.1 Entraînement de l'algorithme

Dans cette section, nous nous intéressons aux valeurs d'états finales obtenues par notre implémentation de l'algorithme *Value Iteration* pour des valeurs de $\delta \in \{1, 0.1, 0.01, 0.005, 0.001\}$ ainsi que le nombre d'itérations k. Pour chaque valeur de δ testée, nous usons comme facteur de réduction $\gamma = 0.9$.

δ	k
1	10
0.1	14
0.01	18
0.005	18
0.001	20

Table 1: Nombre d'itérations de l'algorithme *Value Iteration* en fonction d'un δ donné.

Plus il y a d'itérations, plus nous nous approchons de la convergence. Voici les heatmaps pour $\delta = 1$ et $\delta = 0.001$ affichant les valeurs d'états dans l'environnement stochastique après k itérations:



(a) $\delta = 1$, $k = 10$ (b) $\delta = 0.001$, $k = 20$

Figure 2: Heatmaps générées par l'algorithme *Value Iteration*

Nous nous rendons très vite compte que la valeurs des états les plus éloignés de l'état initial en $(0, 6)$ (en haut à droite) convergent plus rapidement que ceux plus proches de l'état d'origine. Cela s'explique par le fait que toutes les valeurs d'états calculées par *Value Iteration* auront une valeur nulle tant qu'une récompense n'a pas été trouvée. Démontrons cela de manière concise:

Démonstration. En observant l'équation 1, il est clair que la valeur est nulle dans deux cas:

- si $T(s, a, s')$ est nulle. Or cette valeur n'est jamais nulle, il y a toujours une probabilité donnée de passer d'un état s à un état s' suivant.
- si $R(s, a, s') + \gamma V^*(s')$ est nul. Initialement tous les $V(s)$ sont nuls, donc le seul terme qui peut donner une valeur non-nulle à cette expression est la récompense $R(s, a, s')$ (γ est une constante non-nulle).

En somme, les valeurs d'état calculées se propagent à partir des transitions accordant une récompense expliquant ainsi la convergence plus rapide dans ces environs. □

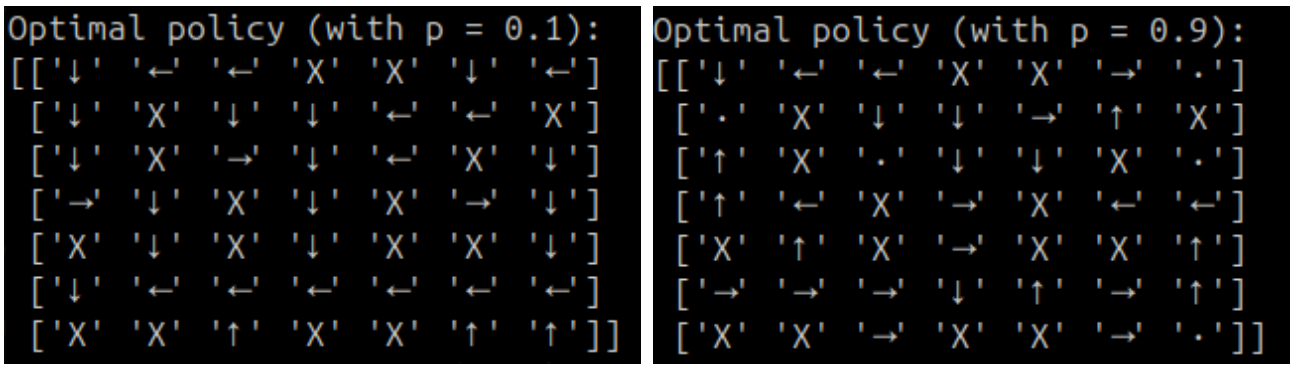
Pour conclure cette section nous souhaitons proposer une analyse du nombre d'itérations requises pour atteindre la convergence. Pour ce faire, nous avons changé la condition d'arrêt de notre algorithme. Dorénavant, il s'arrête dès que la variation entre les valeurs d'états de deux itérations successives est nulle. Ce faisant, nous sommes arrivés à $k = 50$ (et une heatmap exactement identique à celle obtenue avec un $\delta = 0.001$ en $k = 20$ itérations).

Ceci complète donc l'hypothèse fournie dans au cours théorique concernant l'utilité d'un grand nombre d'itérations. En effet, après la moitié du nombre d'itérations requises pour obtenir la convergence nous obtenons déjà des valeurs similaires voire identiques.

2.2 Stratégie optimale

Récupérer la stratégie optimale consiste à récupérer la meilleure action pour chaque état. Cela est aisément faisable en créant une nouvelle matrice $optimal\_policy = np.zeros(shape = env.map.size)$. Ensuite, au lieu d'y placer, pour un état s donné, la meilleure valeur pour cet état, nous y plaçons la meilleure action associée à cette meilleure valeur.

La politique optimale peut changer en fonction du paramètre p de l'environnement (1). Voici nos résultats pour $p = 0.1$ et $p = 0.9$:



(a) $p = 0.1$ (b) $p = 0.9$

Figure 3: Politiques optimales trouvées par l'algorithme *Value Iteration* avec $\delta = 0.01$

Nous observons que l'agent préfère se donner la mort plutôt que d'atteindre une sortie même s'il se trouve à côté de celle-ci. Expliquons cela analytiquement à l'aide d'un exemple concret:

Exemple. Considérons un agent en (0, 5). Trois actions s'offrent à lui: rester, ouest, sud. Aller au sud lui donnerait une récompense de 10. Sauf que cette récompense de 10 est fortement dévaluée par p lorsque nous considérons sud comme action. Lorsque nous considérons ouest par exemple, la récompense sera une fois de plus dévaluée de 90% mais la récompense étant 0, cela import peu. En revanche, comme la formule indique qu'il faut faire la somme de tous les états s' nous repassons sur l'action sud et la récompense de 10 est cette fois ci dévaluée de $\frac{p}{len(available\_actions)}$. Cet élément pèse donc fort dans la somme lorsque l'on considère l'action ouest expliquant ainsi pourquoi l'agent prend cette action au lieu de l'action sud. □

Nous en concluons que les récompenses sont tellement dévaluées que l'agent préfère prendre une action allant à l'opposé de la sortie et se donner la mort.

3 Q-learning

L'algorithme *Q-learning* calcule, pour chaque état et pour chaque action possible à partir de cet état, une valeur réelle. À partir de cette valeur réelle, il détermine une stratégie optimale à suivre en prenant l'action ayant la valeur la plus élevée pour chaque état.

$$Q(s,a) \leftarrow (1-\alpha)\, Q(s,a) + \alpha\big[R(s,a,s') + \gamma\, V(s')\big] \tag{3}$$

Mise à jour de la *q-value* en un état s en prenant une action a, avec $V(s) = \max_{a \in A} Q(s,a)$

Contrairement à *Value Iteration* qui ne considère que des transitions en n'interagissant pas avec l'environnement, *Q-learning* fait un apprentissage *offline* en interagissant directement avec l'environnement. En d'autres termes, si l'agent souhaite déterminer la récompense obtenue en passant de son état s à un nouvel état s' avec une action a il doit exécuter cette action. Donc s'il doit mourir pour savoir que ce n'est pas bon, il le fera.

Une autre différence par rapport au *Value Iteration* est le choix des actions. Le *Q-learning* choisit l'action suivante sur base d'une politique bien déterminée. Il nous a été demandé d'en implémenter deux:

- Exploration ε-greedy: il y a une probabilité ε de choisir une action aléatoire et une probabilité $1 - \varepsilon$ de prendre une action qui maximise $Q(s,a)$. Ceci est formalisé par

l'équation fournie dans les consignes reprise ci-dessous (avec r un nombre uniformément aléatoire entre 0 et 1):

$$\pi(s) = \begin{cases} \arg\max_{a \in A} Q(s,a) & \text{si } r \leq \varepsilon \\ a \sim A & \text{sinon} \end{cases} \tag{4}$$

En fait, ε est une variable qui, plus elle est élevée, plus elle favorise l'exploration et, plus elle est basse, plus elle favorise l'exploitation. L'exploration est lorque l'agent teste une action aléatoire indépendamment des *q-values* déjà calculées pour en découvrir de nouvelles (et potentiellement découvrir d'autres récompenses). L'exploitation fait tout l'inverse. Elle choisit l'action avec la *q-value* la plus élevée. Elle "ne prend pas de risques".

- Exploration softmax: utilise la fonction softmax adaptée à notre environnement. L'équation fournie dans les consignes est la suivante (avec τ la "température"):

$$\pi_a(s) = \frac{e^{\frac{Q(s,a)}{\tau}}}{\sum_{a' \in A} e^{\frac{Q(s,a')}{\tau}}} \tag{5}$$

En termes concrets, pour un état s donné, l'équation va calculer, pour chaque action a disponible, une certaine probabilité. Donc chaque action sera pondérée d'une probabilité p. Nous choisissons ensuite une action aléatoirement en prenant en compte des poids associés.

3.1 Entraînement de l'algorithme

Nous analysons dans cette section les performances de l'agent dans l'environnement en utilisant l'algorithme *Q-learning* sur 20 000 itérations, interactions avec l'environnement en utilisant des paramètres et des politiques différentes.

Les paramètres généraux sont:

- l'environnement: probabilité de prendre une action aléatoire au lieu de celle choisie $p = 0.1$
- l'algorithme: *learning rate*[1] $\alpha = 0.1$ et *discount factor* $\gamma = 0.9$
- $random.seed(0)$

3.1.1 Exploration ε-greedy

Comme exprimé ci-dessus, chaque expérience comporte 20 000 itérations. Nous utilisons un ε différent par expérience. Nous en faisons quatre. Pour les trois premières, nous utilisons $\varepsilon \in \{0, 0.1, 0.5\}$ tandis que pour la dernière nous faisons diminuer ε linéairement de 1 à 0.01 au cours de l'entraînement.

[1]Taux définissant à quel point une ancienne valeur remplace la nouvelle; un taux plus élevé permet à l'agent de s'adapter plus rapidement aux nouvelles valeurs calculées tandis qu'un taux plus bas nous aurons des résultats plus stables mais un apprentissage plus lent.

3.1.2 Exploration softmax

Pour cette politique d'exploration nous utilisons les valeurs de température $\tau \in \{0.01, 1, 10\}$ pour les trois premières expériences. La dernière fait usage d'une fonction diminuant exponentiellement[2] allant de 100 à 0.01 au cours de l'entraînement.

3.1.3 Résultats

Voici les graphiques reprenant les résultats des huit expériences menées:

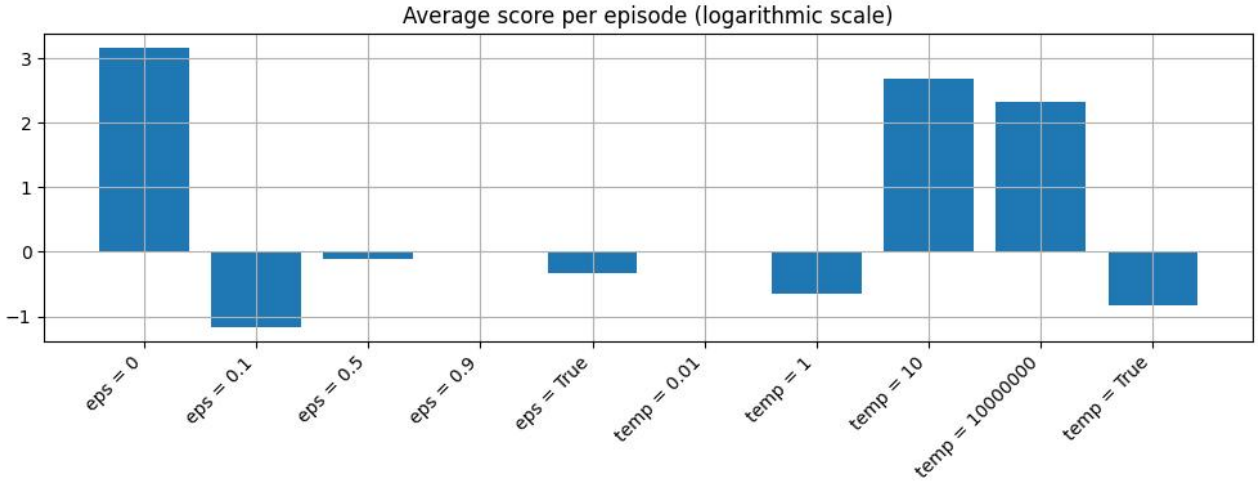


Figure 4: Score moyen par épisode de l'algorithme *Q-learning* avec différentes valeurs par politique d'exploration.

[2]Pour ce faire nous avons utilisé l'expression $f(x) = f(x_0)e^{-kx}$. Nous avons $f(x)$, x et x_0. Il suffit donc de résoudre une équation pour obtenir finalement $k = \frac{4\ln(10)}{20000}$.

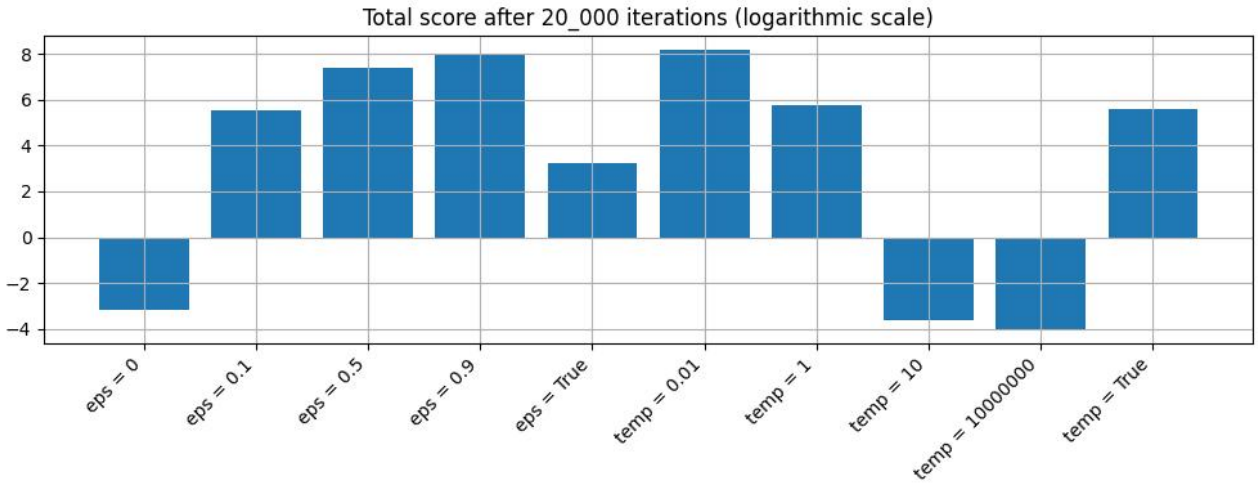


Figure 5: Score total pour une exécution de l'algorithme *Q-learning* avec différentes valeurs par politique d'exploration.

Nous considérons ici un épisode comme toutes les actions prises par l'agent tant qu'il ne meurt pas. En effet, lorsque l'agent meurt, l'environnement est réinitialisé.

Nous observons que lorsque le score moyen est élevé, le score total est très bas voire négatif. Ceci s'explique par l'opposition entre exploitation et exploration. Si l'algorithme ne fait que exploiter (en particulier la sortie la plus proche avec une faible récompense), il aura un score moyen acceptable mais un score total assez mauvais. À l'opposé, une forte exploration implique de nombreuses morts de l'agent expliquant le score moyen faible et un score total élevé[3].

4 Discussion

Cette section vise à répondre aux questionnements initiés à la section 4.3. des consignes.

4.1 Aspect aléatoire

L'algorithme *Value Iteration*, ne comprend pas d'aspect aléatoire. En effet pour calculer la valeur d'un état s, nous considérons chaque action a possible à partir de s pour arriver à différents états s'. Chaque action possède une certaine probabilité d'être effectuée mais comme nous travaillons sur toutes les transitions possibles (la somme de tous les états s'), nous arrivons à "une probabilité totale de 1". Il faut plutôt voir les probabilités associées à chaque action a comme un poids dans l'expression permettant de mieux valuer une transition d'un état s à un état s' avec une action a. Pour concrétiser tout ça, nous avons retiré la seed rendant ainsi notre algorithme en théorie aléatoire. Or, à chaque exécution (avec les mêmes paramètres bien entendu), nous avons obtenus des heatmaps complètement identiques.

[3]Ces résultats se reflètent également dans le nombre de pas moyen effectués par l'agent avant de mourir. Une forte exploitation implique peu de risques et donc beaucoup d'étapes avant de mourir (donnée affichée à l'exécution.)

À l'opposé, l'algorithme *Q-learning* interagit directement avec l'environnement qui possède une probabilité p d'effectuer une action aléatoire au lieu de l'action souhaitée. De plus, les politiques d'exploration ont également un aspect aléatoire. En effet, la politique ε-greedy comprend une probabilité ε de choisir une action aléatoire (encourageant l'exploration) et, la politique softmax qui calcule un poids, une probabilité pour chaque action avant d'en sélectionner une aléatoirement en prenant en compte le poids calculé.

4.2 Stratégies d'exploration

4.2.1 Analytiquement

La stratégie ε-greedy balance son exploration et sa maximisation à l'aide du paramètre ε. En effet, une action aléatoire est choisie avec une probabilité ε, encourageant l'exploration de l'environnement. À l'opposé, il y a une probabilité $1 - \varepsilon$ qu'une action maximisant les *q-values* soit prise, encourageant ainsi l'exploitation.

Concernant *Max-Boltzmann*, le paramètre τ est celui influençant la prise de décision. Lorsque $\tau \to \infty$, nous observons aisément que, dans l'équation 5, nous avons un dividende ayant une valeur tendant vers $e^0 = 1$ et le dénominateur a une valeur tendant vers $\sum_a e^0 = \sum_a 1$. Donc, pour chaque action $a \in A$, nous avons une même probabilité $\frac{1}{len(A)}$ encourageant l'exploration. À l'inverse, quand $\tau \to 0$, les probabilités associées à chaque action a dépendent plus des *q-values* (au numérateurs des exposants), poussant l'algorithme à l'exploitation.

4.2.2 Expérimentalement

L'analyse analytique proposée dans la section 4.2.1 précédente est appuyée par les résultats[4] obtenus avec les paramètres suivants:

- random.seed = 0
- $\gamma = 0.9$
- $\alpha = 0.1$
- nombre d'étapes = 20_000

ε-greedy explore bien comme prévu plus ε est faible. Plus ε augmente, moins il explore. Ceci est visible sur la heatmap avec $\varepsilon = 0.9$: les cases à côté de la première sortie (la plus proche) sont jaunes (valeur élevée) tandis que la sortie la plus lointaine voit ses cases voisines vierges de valeurs; l'exploration n'a pas été suffisament profonde. Finalement, concernant l'expérience avec ε diminuant linéairement, nous devrions avoir une heatmap relativement bien explorée (ε élevé au début de l'algorithme) avec une exploitation plus poussée vers la fin. Cependant, avec 20_000 itérations, l'algorithme ne trouve pas la sortie la plus éloignée. Nous avons alors essayé avec 50_000 itérations et nous voyons immédiatement que la sortie lointaine est découverte et a une case très jaune à proximité. La conclusion est donc la suivante: sur plus d'itérations,

[4]Afin de ne pas surcharger ce document d'avantage, nous analysons les résultats sans ajouter les heatmaps. Cependant, nous proposons les paramètres d'entrée afin de permettre au lecteur de reproduire les expériences. De plus, toutes les heatmaps sont disponibles dans le directory *./graphs*.

l'algorithme a eu plus de temps pour explorer, plus d'itérations avec un ε relativement élevé pour finalement bien exploiter les cases aux *q-values* les plus élevées.

La stratégie *Max-Boltzmann*, comme ε-greedy colle à la théorie: plus $\tau \to 0$, moins il y a d'exploration. Cependant, nous observons que avec une valeur de τ favorisant l'exploration ou l'exploitation, des *q-values* sont calculées de manière à peu près équitable partout dans l'environnement (sauf pour les valeurs extrêmes comme par exemple $\tau = 0.01$ où les *q-values* ne s'étendent pas jusqu'à la sortie éloignée).

4.2.3 Conclusion

Ces analyses permettent donc de répondre à la question concernant l'équilibre des politiques d'exploration. La stratégie d'exploration *Max-Boltzmann* semble atteindre un meilleur équilibre entre exploration et exploitation. Chaque action possède une probabilité propre d'être sélectionnée. Donc, même si une action possède une faible probabilité, elle peut se voir être choisie tout de même. En d'autres termes, les actions ayant une *q-value* plus élevées sont favorisées mais pas choisies exclusivement. De plus, ces calculs de probabilité sont pondérés par le paramètre τ permettant de diriger, de choisir une approche tendant plus vers l'exploitation ou l'exploration (mais jamais un des deux exclusivement).

En effet, à son opposé, la stratégie ε-greedy est binaire. Il y a une probabilité ε de prendre une action aléatoire. Donc, avec ε proche de 1, nous avons un algorithme qui explore majoritairement en négligeant l'exploitation (et inversément pour ε proche de 0). On ne retrouve donc pas la même stabilité que chez *Max-Boltzmann*.

4.3 Effet du discount factor γ

Nous pouvons exprimer le *discount factor* comme un facteur qui diminue l'importance des valeurs des états futurs s' atteints à partir de s en effectuant l'action a.

Dans le cas de notre programme, comme nous l'avons déjà expliqué, les valeurs d'état se "propagent" à partir des transitions d'un état s à un état s' avec une action a qui octroient une récompense. Diminuer γ aura pour effet de réduire ce phénomène de propagation et diminue donc les valeurs des états en général. De ce fait, les cases des heatmaps (pour *Value Iteration* et *Q-learning*) auront de plus petites valeurs et donc des cases avec des couleurs plus faibles. Pour des valeurs de γ particulièrement faibles (ex: $\gamma = 0.1$) nous observons que seules les sorties sont claires, en particulier celles avec une récompense plus élevée si celle-ci est atteinte (exploration suffisante requise).

4.4 Effet du taux d'apprentissage α

Le taux d'apprentissage balance les valeurs *q-values* précédemment calculées et les nouvelles (valeur d'état s' et la récompense obtenue en arrivant à cet état s'). Nous observons à l'équation 3 que, plus α est élevé, plus les nouvelles valeurs auront un poids important dans le calcul du nouveau poids et moins l'ancienne valeur d'état aura de poids (et inversément pour α faible).

En particulier, nous observons que, en comparant un $\alpha = 0.9$ et $\alpha = 0.1$ sur la stratégie ε-greedy avec $\varepsilon = 0.1$ (exploration très faible), les *q-values* sont mieux réparties. En effet, avec $\alpha = 0.1$, uniquement les cases proches des sorties sont colorées et possèdent donc une valeur non-nulle ou élevée.

Nous en tirons que, plus α est élevé, plus les *q-values* précédemment calculées sont écrasées. Ceci est justifié par notre exemple ci-dessus. Avec $\varepsilon = 0$, ε-greedy n'explore que peu et exploite surtout. Avec $\alpha = 0.1$ le chemin menant aux sorties possède des *q-values* quasi nulles car, l'algorithme prend toujours la même action et ne calcule que rarement les *q-values* éloignées des sorties. Avec une propagation plus importante, le chemin est donc plus éclairé car, même

pour un passage unique par un chemin, même pour une seule exploration, les valeurs précédentes ne sont pratiquement pas prises en compte; uniquement la récompense compte.

5 Conclusion

Ce troisième projet d'Intelligence Artificielle nous a permis une fois de plus d'appliquer un concept vu en cours, l'analyser et conclure. Nous avons pu peser les avantages et inconvénients de chaque algorithme d'apprentissage par renforcement implémenté, à savoir le *Value Iteration* et le *Q-learning*. L'analyse des heatmaps générées nous a permis de visualiser les algorithmes de façon claire permettant ainsi une compréhension accrue de ce qui nous a été demandé.