Robotique et apprentissage, TME 2

ABITBOL Yossef, 3804139 et KOSTADINOVIC Nikola, 3801695

February 2022

RENFORCEMENT LEARNING

Sommaire

1 Introduction

Ce TME est divisé en deux parties, la première correspondant a de la **programmation dynamique** dans lequel nous étudierons les algorithmes d'itération de valeur et d'itération de politique dans un environnement de labyrinthe. La seconde, focalisé sur **l'apprentissage par renforcement tabulaire** va nous permettre d'étudier les algorithmes de base: TD learning, q-learning et sarsa. Nous étudierons également deux stratégies d'exploration de base : epsilon-greedy et softmax.

2 Programmation Dynamique

2.1 StudyPart 1: Experimental comparisons

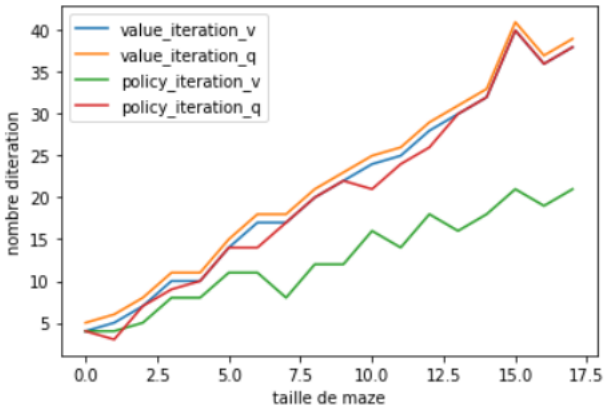
We will now compare the efficiency of the various dynamic programming methods using either the V or the Q functions.
In all your dymanic programming functions, add code to count the number of iterations and the number of elementary V or Q updates. Use the provided mazemdp.Chrono class to measure the time taken. You may generate various mazes of various sizes to figure out the influence of the maze topology.
Build a table where you compare the various dymanic programming functions in terms of iterations, elementary operations and time taken.

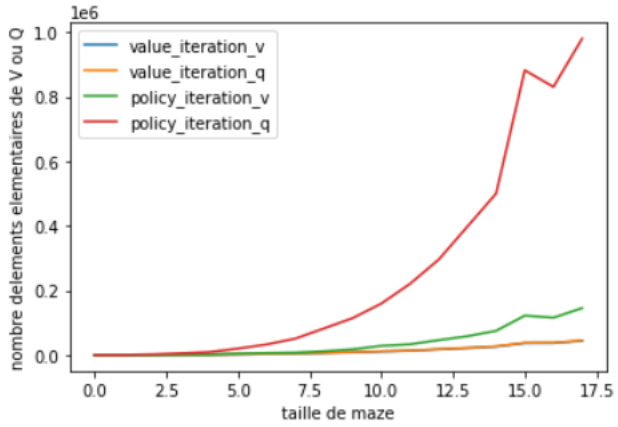
Pour répondre a cette question, on ajoute 2 variables pour incrémenter le nombre d'itérations mais aussi le nombre d'éléments élémentaires. Ensuite, on récupère ces nombres afin de les comparer.

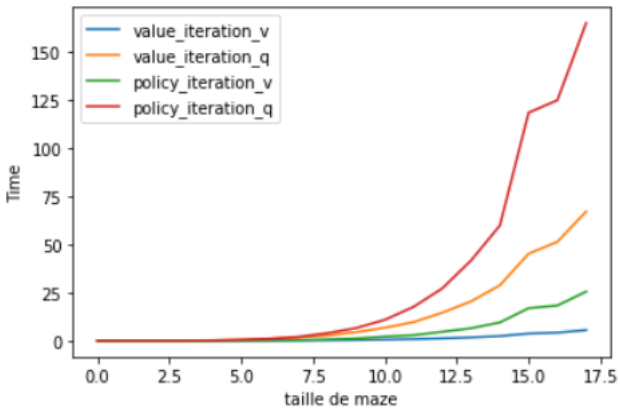
```
1
2  from mazemdp import chrono
3  import pandas as pd
4  import time
5
6  def plot_convergence_vi_pi(m, render):
7      liste_chrono = []
8      # c = chrono.Chrono()
9      start = time.time()
10     v, v_list1 ,iterations1 , elementaires1 = value_iteration_v(m, render)
11     # c.stop()
12     fin = time.time()
13     liste_chrono.append(fin-start)
14
15     start = time.time()
16     q, q_list1 ,iterations2 , elementaires2 = value_iteration_q(m, render)
17     fin = time.time()
18     liste_chrono.append(fin-start)
19
20     start = time.time()
21     v, v_list2 ,iterations3 , elementaires3 = policy_iteration_v(m, render)
22     fin = time.time()
23     liste_chrono.append(fin-start)
24
25     c = chrono.Chrono()
26     start = time.time()
27     q, q_list2 ,iterations4 , elementaires4 = policy_iteration_q(m, render)
28     fin = time.time()
29     liste_chrono.append(fin-start)
30
31     df = pd.DataFrame({'value_iteration_v' : [iterations1 ,elementaires1 ,liste_chrono[0]],
32                        'value_iteration_q' : [iterations2 ,elementaires2 ,liste_chrono[1]],
33                        'policy_iteration_v' : [iterations3 ,elementaires3 ,liste_chrono[2]],
34                        'policy_iteration_q' : [iterations4 ,elementaires4 ,liste_chrono[3]]})
35     renom = {0:'nombres diterations',1:'nombre elementaires',2:'time (ms) '}
36     df.rename(renom,axis=0,inplace=True)
```

```
    plt.plot(range(len(v_list1)), v_list1, label='value_iteration_v')
    plt.plot(range(len(q_list1)), q_list1, label='value_iteration_q')
    plt.plot(range(len(v_list2)), v_list2, label='policy_iteration_v')
    plt.plot(range(len(q_list2)), q_list2, label='policy_iteration_q')

    plt.xlabel('Number of episodes')
    plt.ylabel('Norm of V or Q value')
    plt.legend(loc='upper right')
    plt.savefig("comparison_DP.png")
    plt.title("Comparison of convergence rates")
    plt.show()

    return df
```

note: On a préféré utiliser "time.time()" au lieu de chrono (après vérification des mêmes résultats) pour pouvoir stocker dans une variable et l'afficher dans notre DataFrame.
On génère les courbes pour une taille allant de 2 a 20 afin d'avoir une idée.

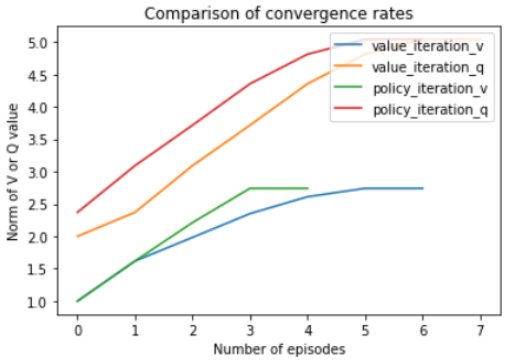






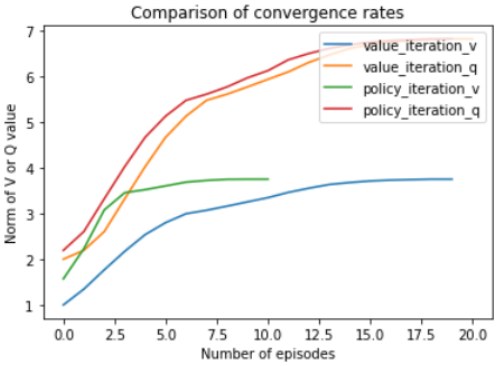
En exécutant ce code avec 3 MDP de tailles différentes, 4, 10 et 20 on obtient les 3 courbes suivantes:

```
mdp = create_random_maze(4, 4, 0.2)
plot_convergence_vi_pi(mdp, False)
```



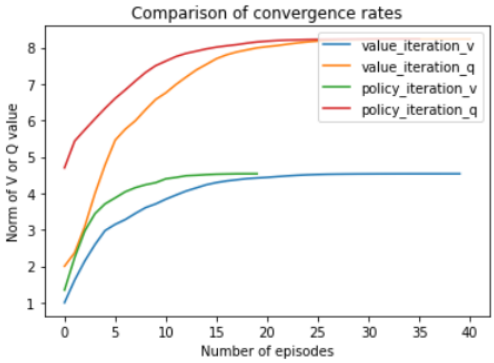
	value_iteration_v	value_iteration_q	policy_iteration_v	policy_iteration_q
nombres diterations	7.000000	8.000000	5.000000	7.000000
nombre elementaires de Q ou V	364.000000	416.000000	325.000000	1968.000000
time (ms)	0.008243	0.028154	0.006276	0.022901

```
mdp = create_random_maze(10, 10, 0.2)
plot_convergence_vi_pi(mdp, False)
```



	value_iteration_v	value_iteration_q	policy_iteration_v	policy_iteration_q
nombres diterations	20.000000	21.000000	11.000000	20.000000
nombre elementaires de Q ou V	6400.000000	6720.000000	13440.000000	80264.000000
time (ms)	0.309741	2.717892	0.874352	4.540309

```
[108] mdp = create_random_maze(20, 20, 0.2)
      plot_convergence_vi_pi(mdp, False)
```



	value_iteration_v	value_iteration_q	policy_iteration_v	policy_iteration_q
nombres diterations	40.000000	41.000000	20.000000	3.600000e+01
nombre elementaires de Q ou V	51200.000000	52480.000000	161280.000000	1.173920e+06
time (ms)	9.306506	97.779086	36.996621	2.586315e+02

On remarque que peu importe la taille du mdp, **l'ordre des courbes est toujours le même**, Q-value function avec une norme plus élevé que V-value function. De plus, les 3 tests avec les tailles différentes dégagent les mêmes conclusions, le nombre d'itération est le **moins élevé pour policy_iteration_v** suivi de policy_iteration_q et value_iteration_v et enfin le plus élevé avec value_iteration_q.

En ce qui concerne le temps, plus on augmente la taille plus le temps est élevé et plus l'écart se creuse comme on peut le voir avec une taille de 20, **value_iteration_v est toujours le plus rapide** avec 27s d'ecart avec policy_iteration_v qui peut etre expliqué par le fait que les mis a jour de Policy iteration v sont plus couteuses, on évalue la politique puis on l'ameliore alors que pour value iteration on améliore directement la politique avec les valeurs venant d'être calculées, 88s d'écart avec value_iteration_q et enfin 250s avec policy_iteration_q.

Aussi, le nombre d'éléments élémentaires de Q ou V est minimum chez les value_iteration comparé aux policy_iteration.

Pour finir, on regarde sur les courbes tracés, plus la taille augmente, plus le nombre de épisode pour atteindre convergence augmente. Prenons l'exemple de **la premiere converge atteinte qui est policy_iteration_v (courbe en verte)**. Pour une taille de 4*4 la convergence est atteinte a un nombre d'épisode de 3 puis a 10*10 cette valeur passe a environ 6 et enfin pour 20*20 on atteint convergence a 12.

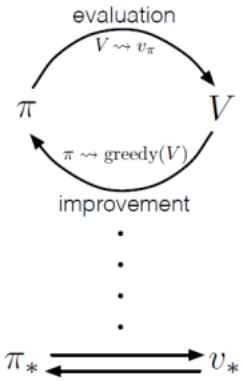
On conclut donc que la méthode qui converge avec le moins du nombre d'épisode, qui a le moins d'itérations et qui est la 2e plus rapide est **policy iteration v**. Les Q-Values sont un excellent moyen de rendre les actions explicites afin que vous puissiez traiter les problèmes où la fonction de transition n'est pas disponible (model-free). Cependant, lorsque votre espace d'action est grand, les choses ne sont pas si agréables et les Q-values ne sont pas si pratiques comme avec un grand nombre d'actions ou même à des espaces d'action continus.
Du point de vue de l'échantillonnage, la dimensionnalité de Q(s,a) est supérieure à V(s), il peut donc être **plus difficile d'obtenir suffisamment d'échantillons (s,a) par rapport à (s).**
Il est **plus simple** de prendre des décisions sur des actions basées sur des Q-value que sur des V-values car pour choisir l'action en fonction de l'état, l'agent n'a qu'à calculer la Q-value pour toutes les actions disponibles en utilisant l'état actuel et choisir l'action avec la plus grande Q-value. Alors

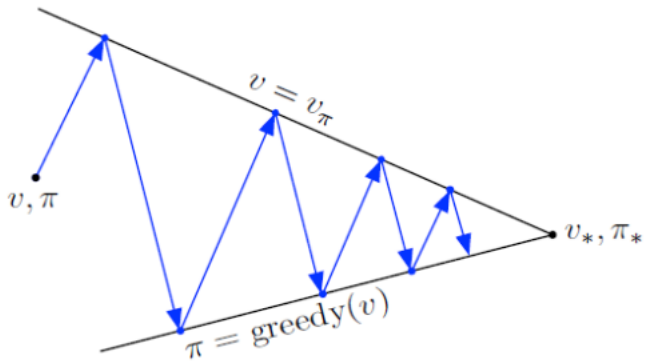
que les V-value, l'agent doit connaître les valeurs et les probabilités de transitions. Ce qui est rarement le cas à l'avance, l'agent estime donc les probabilités de transition pour chaque paire d'action et d'état.

2.2 StudyPart 2: Generalized Policy Iteration

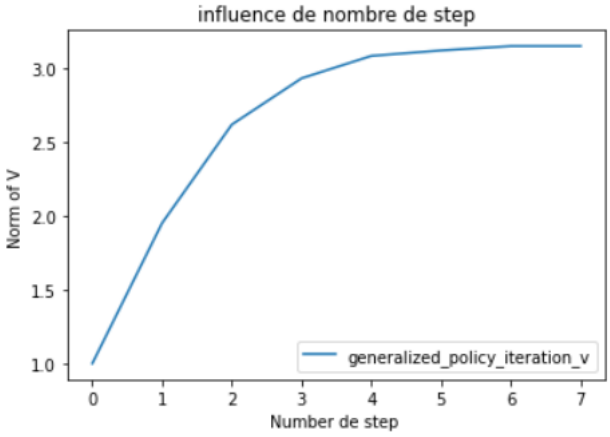
Code the generalized policy iteration algorithm and study the influence of the number of evaluation steps between each improvement step

---

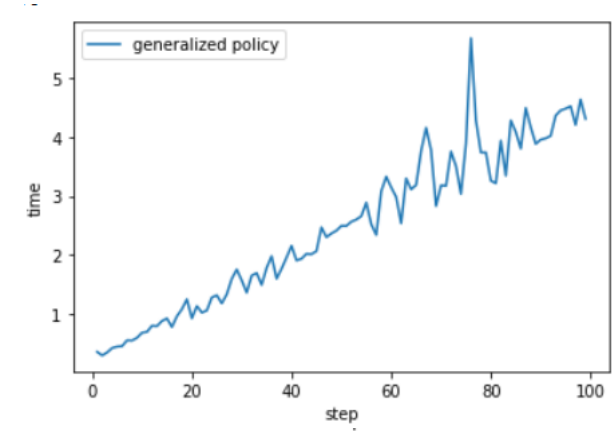


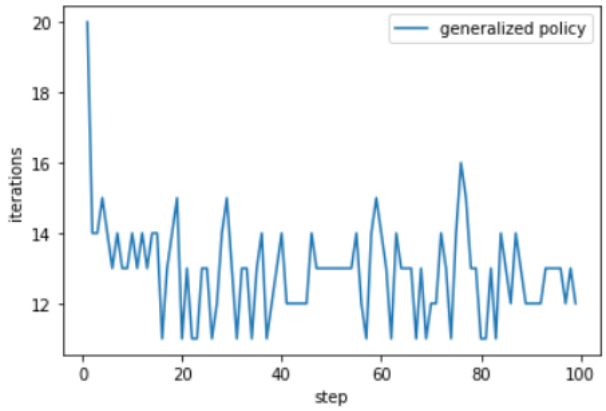


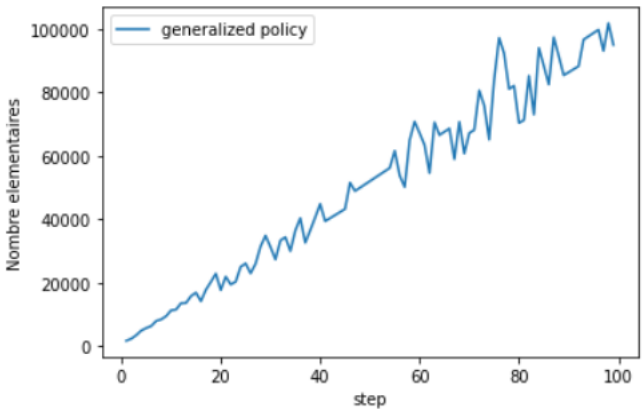
Cette image représente comment on doit coder generalized policy itération en réutilisant les fonctions déjà implémenté auparavant et on modifie en faisais un for. (le code est long donc on na préféré ne pas le mettre dans le rapport). Cependant, on obtient en faisant les modifications appropriés la courbe suivante:
(On crée un random maze de taille 5*5)



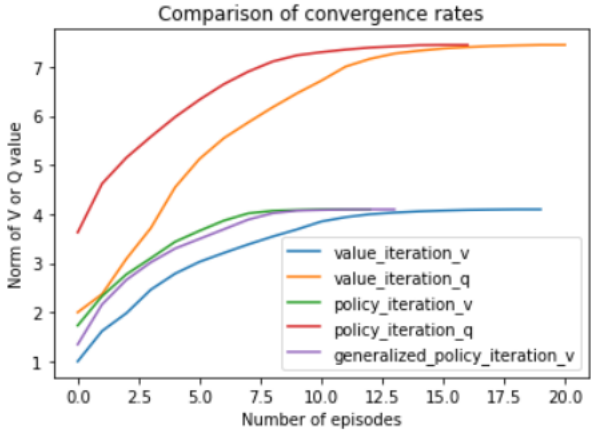
On remarque qu'il nous a fallu 7 steps pour converger.
Enfin, on compare le temps, le nombre d'itérations et le nombre d'éléments élémentaires avec les autres fonctions. Pour cela on fait varier le nombre de steps avec un maze de 10*10.







On remarque que plus on augmente le nombre de steps plus le temps et le nombre d'éléments élémentaires augmentent contrairement aux nombre d'itérations qui reste stable vers les 13.
Pour le time, de 0 a 10 steps on a a peu près le meme temps que valu itération v et policy itération v, entre 50 et 60 steps le meme temps que value itération q et enfin il faut plus de 90 steps pour atteindre le meme temps que policy itération q.
Pour le nombre d'itérations, on stagne autour de 13, supérieur a policy iteration v mais inférieur aux autres.
Enfin, pour le nombre d'éléments élémentaires de V, plus de 80 steps pour dépasser 80000 et entre 0 et 10 step pour avoir un nombre entre 0 et 15000.
La dernière comparaison se fait au niveau de la convergence.



La courbe est presque la meme que policy itération v Pour un nombre de step de 20 et une maze de 10*10.

3 Apprentissage par Renforcement Tabulaire

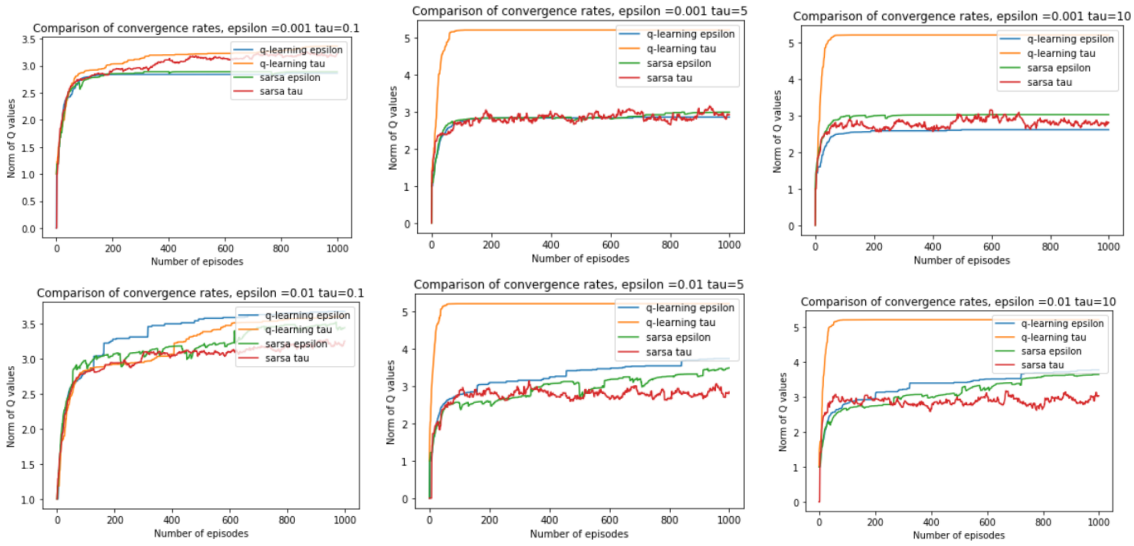
3.1 StudyPart 1: Impact of ϵ and τ on q-learning and sarsa

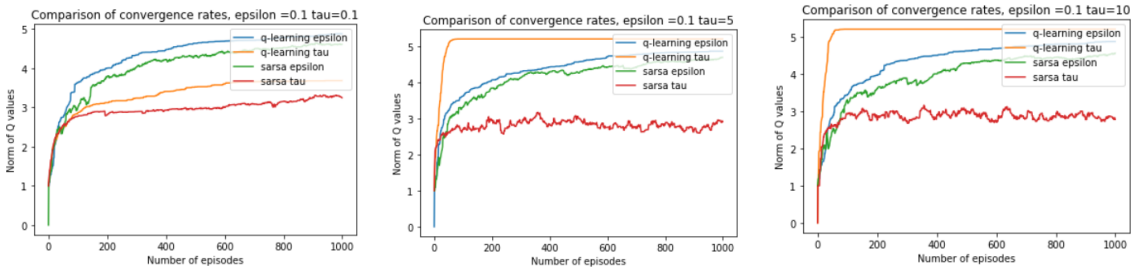
Compare the number of steps needed by q-learning and sarsa to converge on a given MDP using the softmax and -greedy exploration strategies. To figure out, you can use the provided plot_ql_sarsa(m, epsilon, tau, nb_episodes, timeout, alpha, render) function below with various values for (e.g. 0.001, 0.01, 0.1) and τ (e.g. 0.1, 5, 10) and comment the obtained curves. Other visualizations are welcome.

Ici, il suffit juste de faire varier les paramètres de tau et de epsilon pour obtenir les courbes correspondantes comme cela:

```
for i in [0.001 , 0.01, 0.1]:
  for j in [0.1,5,10]:
    print("epsilon = "+str(i)+" et tau = "+str(j))
    plot_ql_sarsa(mdp, i, j, 1000, 50, 0.5, False)
```

Ainsi, on a les 9 courbes suivants:

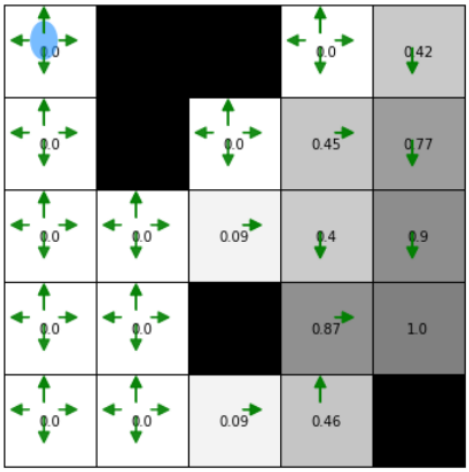




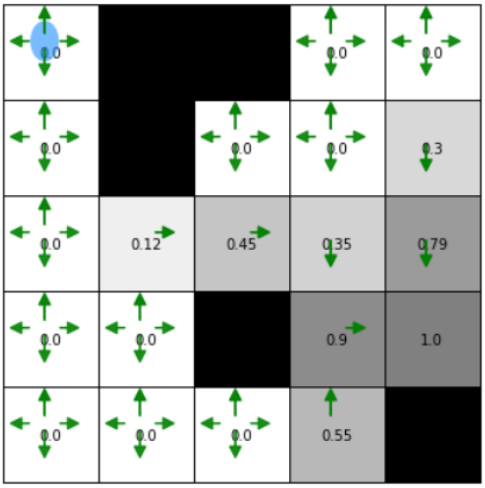
Premièrement, on observe que avec $\epsilon = 0.001$ et $\tau = 0.1$ on a a peu près les même performances pour tous les algorithmes, avec une norme de Q entre 2,8 et 3,3, ce qui signifie que lorsque τ est faible Q-learning et Sarsa performent a peu près de la même manière sauf que lorsqu'on augmente ϵ l'écart de la norme de Q augmente.
Ensuite, plus on augmente τ, plus on observe que la convergence se fait plus rapidement, comme par exemple pour q-learning tau on peut voir la différences de convergence entre la courbe a $\epsilon = 0.1$ et $\tau = 0.1$ et la courbe a $\epsilon = 0.1$ et $\tau = 5$. Aussi, Q-learning va plus apprendre parce que un grand tau signifie moins d'écarts entre les Q-values. Il se retrouve sur des cases qu'il na jamais ou peu visité et effectue de nouvelles actions, ce qui lui permet de mieux comprendre le labyrinthe et surtout quoi faire.
De plus, d'après les 9 figures, on remarque que softmax converge plus vite que e-greedy. Aussi, plus on diminue epsilon et plus vite on convergera (100 épisodes), ceci est bien montré avec les 3 figures du milieu ($\tau = 5$) et lorsque epsilon devient grand,on observe que Q-learning est toujours en train d'augmenter mais doucement donc sa convergence n'est pas très clair.

Pour conclure, la methode SARSA a une convergence plus rapide si nous utilisons softmax method. Plus on diminue epsilon et plus on convergera rapidement et lorsque la Q-learning converge sa norme de Q ne changera pas contrairement a SARSA.
On compare les 2 affichages entre Q-learning et Sarsa en fonction de epsilon.

```
epsilon = 0.01
q, q_list = q_learning_eps(mdp, epsilon)
```

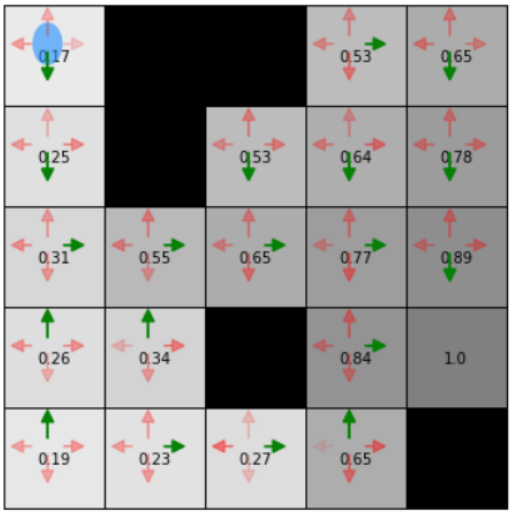



```
epsilon = 0.01
q, q_list = sarsa_eps(mdp, epsilon)
```

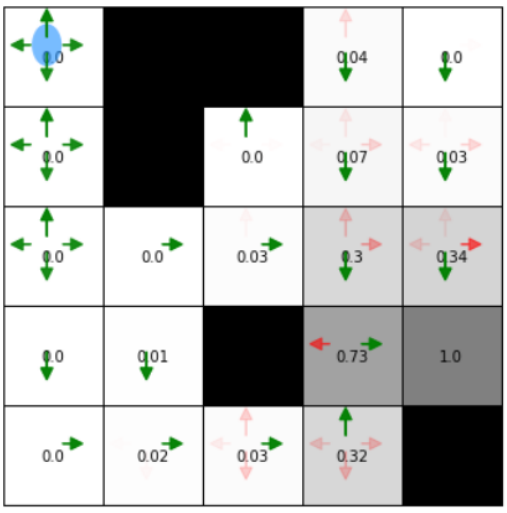


On remarque que Q-learning a observé plus de cases que Sarsa (2 de plus).
On compare les 2 affichages entre Q-learning et Sarsa en fonction de tau.

```
tau = 5
q, q_list = q_learning_soft(mdp, tau)
```




```
tau = 5
q, q_list = sarsa_soft(mdp, tau)
```



Ici, on observe que toutes les cases de Q-learning ont été parcouru et chaque case a une probabilité alors que pour Sarsa, certaines cases n'ont pas de probabilité ou des probabilités de 0. En ce qui concerne les flèches, les cases du graphique de gauche pointe toujours vers une case suivante pour pouvoir se créer un chemin jusqu'à la case finale comme pour le graphique de droite sauf que certaines cases pointent vers le mur ou avec des flèches rouges.

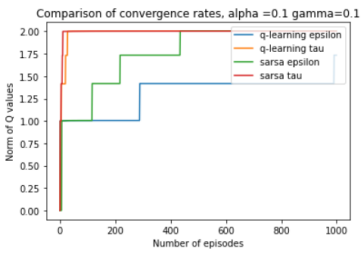
3.2 StudyPart 2: Effect of hyper-parameters

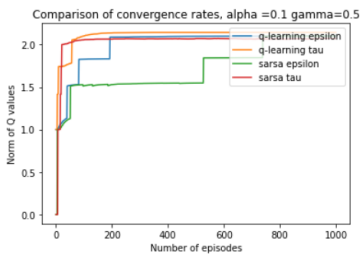
The other two hyper-parameters of q-learning and sarsa are α , and γ . By varying the values of these hyper-parameters and watching the learning process and behavior of the agent, explain their impact on the algorithm. Using additional plotting functions is also welcome.

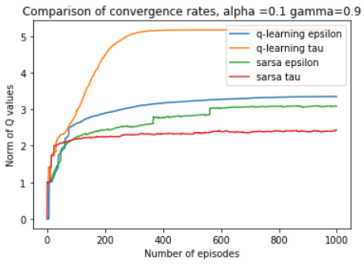
Dans cette partie, on se focalise sur les hyperparamètres γ et α. Pour faire cela, on exécute le code suivant :

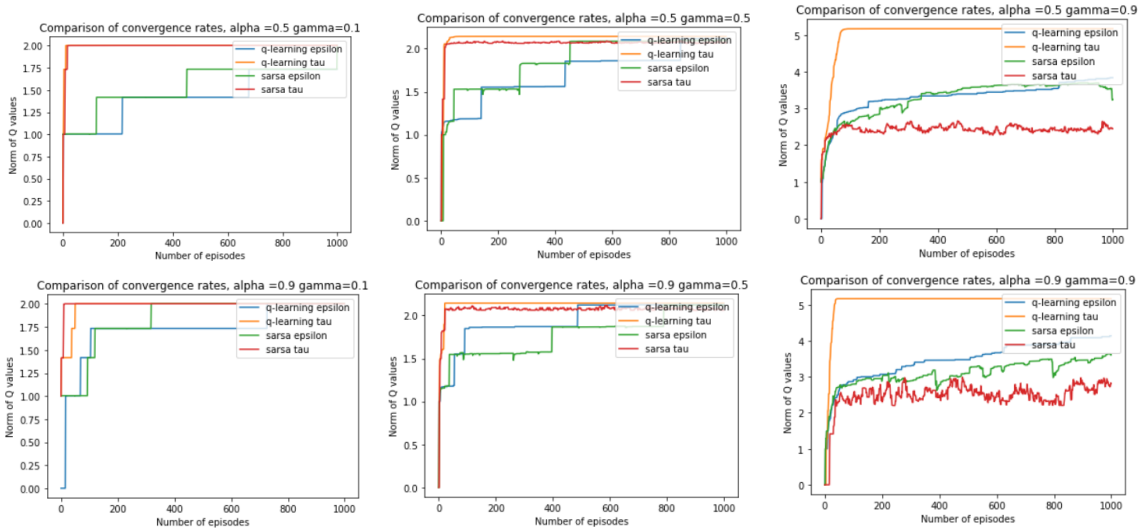
```
for i in [0.1,0.5,0.9]:
  for j in [0.1,0.5,0.9]:
      mdp.gamma = j
      plot_ql_sarsa(mdp, 0.01, 5, 1000, 50, i, False)
```

Comme on peut voir sur le code plus haut, on fixe epsilon a 0.01 et tau a 5. On fait varier alpha [0.1,0.5,0.9] et gamma [0.1,0.5,0.9].









Tout d'abord , nous constatons que lorsque nous augmentons la valeur du learning rate α, plus la valeur va osciller autour du point minimum. Ce phénomène est du au fait que parfois nous avons trouvé la politique optimale mais l'algorithme fait varier cette valeur aussi tot en changeant la politique. Au contraire plus la valeur de ce learning rate est petite, plus le temps mis par l'algorithme pour trouver les véritables valeurs de la Q-table (celles qui permettent à tout agent la connaissant de trouver la case finale en effectuant le moins de déplacements possibles à partir de n'importe quelle position) est long.

De plus, nous voyons que le discount factor γ a une influence sur le MDP. En effet, nous remarquons pour la méthode Q-learning que plus on augmente la valeur du discount factor, plus la norme de Q augmente. Cependant la vitesse de convergence ne varie pas. Concernant la méthode Sarsa, nous pouvons voir que plus on augmente γ, plus la valeur de la norme de Q oscille. D'apres le cours, plus le discount factor est proche de 0, plus l'agent sera sensible au récompense immédiate, au contraire si γ est proche de 1, alors l'agent sera sensible aux récompenses lointaines. Plus γ est petit plus l'agent met du temps à trouver une politique complète et optimale. Lorsque celui-ci est trop faible, l'erreur de prédiction faite à chaque étape sur la valeur qui aurait du être attribuée à chaque case (TD error) est faible. Ainsi, les mises à jour des valeurs de la Q-table se font très lentement. Ici, les meilleurs résultats ont été obtenus pour $\gamma = 0.9$.