Résolution du Problème de Labyrinthe

Donia Tekaya

Étudiante en Master 1 BDIA

et

Mohamed Elyes Maalel

Étudiant en Master

6 mai 2024

Table des matières

Table des figures

Introduction Générale

Le projet aborde le problème du labyrinthe dans le but de découvrir un chemin optimal permettant de surmonter les obstacles, en utilisant l'apprentissage par renforcement. Ce problème est pertinent en intelligence artificielle car il illustre de manière simplifiée les défis de navigation et de prise de décision rencontrés dans des environnements réels. Nous nous focalisons sur l'algorithme de Q-learning, en adoptant une stratégie d'action ϵ-greedy, où le facteur de dépréciation est fixé à 0,9. Les récompenses sont attribuées en fonction des résultats du chemin suivi : une case sortie vaut 100 points, un mur vaut -10 points, et les autres cases valent -1 point.

L'objectif de ce projet est d'appliquer cet algorithme au problème du labyrinthe, d'illustrer graphiquement le chemin trouvé, et d'explorer l'influence des paramètres ϵ (coefficient d'apprentissage) et γ (facteur de dépréciation) sur les performances. Les variations des paramètres seront analysées pour évaluer l'impact sur le nombre de cycles nécessaires pour trouver la sortie. De plus, nous étudierons les performances de l'algorithme lorsque la taille du labyrinthe est modifiée (grilles 20x20 et 30x30). Les résultats obtenus seront comparés et interprétés graphiquement pour offrir une meilleure compréhension des stratégies optimales.

CHAPITRE 1

concept de l'apprentissage par renforcement

1.1 Introduction à l'Apprentissage par Renforcement

L'apprentissage par renforcement (Reinforcement Learning, RL) est une branche de l'intelligence artificielle qui traite de la manière dont les agents logiciels devraient prendre des décisions dans un environnement afin de maximiser une notion de récompense cumulative. Contrairement à d'autres types d'apprentissage automatique, l'apprentissage par renforcement est caractérisé par une interaction entre un agent et son environnement, dans lequel l'agent apprend à atteindre un objectif en essayant différentes actions et en observant les résultats de ces actions.

1.1.1 Les termes fondamentaux de l'apprentissage par renforcement :

Dans le cadre de l'apprentissage par renforcement, les termes suivants sont fondamentaux pour comprendre comment les agents interagissent avec leur environnement pour apprendre des politiques efficaces :

Agent :

FIGURE1.1 – Agent

L'agent est l'entité qui prend des décisions dans un environnement. Il effectue des actions en se basant sur son observation actuelle de l'état de l'environnement ou sur une politique prédéfinie. L'objectif de l'agent est de maximiser la

somme des récompenses qu'il reçoit de l'environnement. Dans un jeu de plateau, par exemple, l'agent pourrait être le joueur qui décide des mouvements à faire.

Environnement : L'environnement est le monde extérieur avec lequel l'agent interagit et sur lequel il n'a pas un contrôle total. Il peut être tout système ou cadre qui répond aux actions de l'agent par des changements d'états et des récompenses. L'environnement détermine les états et fournit des récompenses ou des pénalités en réponse aux actions de l'agent.

État : Un état est une description de la situation actuelle de l'environnement dans laquelle se trouve l'agent. Chaque action que l'agent prend résulte dans un nouvel état, qui peut être le même ou différent. Les états fournissent le contexte nécessaire à l'agent pour prendre des décisions. Par exemple, dans un jeu d'échecs, un état serait la configuration actuelle des pièces sur l'échiquier.

Action : Une action est une intervention spécifique que l'agent effectue dans un état donné. L'ensemble des actions disponibles peut varier d'un état à l'autre. Les actions de l'agent influencent l'état suivant de l'environnement et les récompenses ultérieures que l'agent reçoit.

Récompenses à l'instant t : Une récompense est un feedback immédiat fourni par l'environnement en réponse à une action prise par l'agent. La récompense à l'instant t, notée souvent R_t, est un indicateur de la valeur de l'action prise : elle peut être positive (encourageant l'action) ou négative (décourageant l'action). L'agent cherche à maximiser la somme totale des récompenses obtenues au cours du temps, ce qui peut impliquer des compromis entre les bénéfices immédiats et les récompenses futures.

Ces concepts forment la base de l'interaction dans l'apprentissage par renforcement, où l'agent apprend à naviguer dans l'environnement de manière à maximiser ses récompenses cumulatives.

1.1.2 Processus d'Apprentissage :

Le processus d'apprentissage dans le cadre de l'apprentissage par renforcement (RL) implique une série de décisions que l'agent doit prendre pour apprendre de son environnement de manière efficace. Ce processus est essentiellement itératif, où chaque étape ou décision contribue à l'amélioration progressive des connaissances de l'agent sur l'environnement. Un aspect central de ce processus est la dualité entre exploration et exploitation, deux stratégies qui doivent être équilibrées pour permettre un apprentissage optimal. Voici une explication plus détaillée de ces concepts :

1.1.2.1 Exploration et Exploitation

Exploration :

But : Découvrir de nouvelles connaissances sur l'environnement.

Comment : L'agent prend des actions au hasard, indépendamment de la politique actuelle. Cela permet à l'agent de visiter et de tester des états qui ne sont pas fréquemment explorés dans le cadre de la politique actuelle.

Pourquoi : Sans exploration suffisante, l'agent risque de ne jamais découvrir des actions qui pourraient être plus avantageuses que celles déjà connues. L'explora-

tion est cruciale, particulièrement dans les étapes initiales de l'apprentissage ou dans des environnements très dynamiques où les conditions changent et où de nouvelles stratégies peuvent devenir nécessaires.

Risque : Trop d'exploration peut conduire à des décisions inefficaces, car l'agent peut continuer à choisir des actions suboptimales au détriment de l'accumulation de récompenses immédiates.

Exploitation :

But : Maximiser les récompenses basées sur les connaissances actuelles.

Comment : L'agent choisit les actions qui, selon sa connaissance actuelle (souvent représentée par une table de valeurs Q ou une politique), semblent offrir la meilleure récompense.

Pourquoi : L'exploitation permet à l'agent d'accumuler des récompenses de manière efficace en utilisant les meilleures stratégies qu'il a apprises jusqu'à présent.

Risque : Si l'agent exploite exclusivement sans continuer à explorer, il peut rester piégé dans des comportements localement optimaux et ne jamais découvrir d'autres actions qui pourraient être plus bénéfiques à long terme.

1.1.3 Description de l'Algorithme Q-learning

Le Q-learning cherche à apprendre une fonction de valeur d'action, $Q(s, a)$, qui donne une estimation de la valeur totale des récompenses futures obtenues en prenant l'action a dans l'état s et en suivant ensuite la meilleure politique. Le but est de construire une table Q qui guide directement les décisions optimales.

Étapes de l'Algorithme Q-learning

```
procedure QLEARNING
    ∀s∀aQ(s, a) ← 0
    for n ← 1, nbCycles do                          ▷ nbCycles d'apprentissage
        λ ← 1 ; ϵ ← 1 ;
        etatCourant ← etatInitial
        for i ← 1, nbMaxActions do                  ▷ max supposé d'actions à tester
            s ← etatCourant
            nb ← random(0, 1)
            if (nb < ϵ) then
                a ← randomAction(s)                 ▷ choix aléatoire d'une action à partir de s
            else
                a ← argMax_a'(Q(s, a'))             ▷ choix de l'action de s avec Q maximum
            end if
            s' ← a(s)                               ▷ s' est l'état d'arrivé après exécution de a en s
            Q(s, a) ← λ × (r + γ × max_a'(Q(s', a')) + (1 − λ) × Q(s, a)
            λ ← 0.99 × λ                            ▷ décrémenter les coefficients
            ϵ ← 0.99 × ϵ
            if (s' = etatFinal) then
                Sortie boucle i
            end if
        end for
    end for
```

FIGURE1.2 – l'algorithme de Qlearning

Initialisation :

Initialisez la table Q à des valeurs initiales pour chaque paire état-action. Ces valeurs peuvent être mises à zéro ou à toute petite valeur aléatoire, selon la méthode préférée.

Boucles d'Apprentissage :

Pour chaque épisode d'apprentissage (un épisode commence généralement dans un état initial et se termine lorsque l'état terminal est atteint ou après un nombre maximum d'étapes) :

Choix d'une Action (Stratégie $\epsilon - greedy$) :

Sélectionnez une action a pour l'état courant s en utilisant une politique dérivée

de Q (par exemple, $\epsilon - greedy$). Avec une probabilité ϵ, choisissez une action aléatoire (exploration), et avec une probabilité $1-\epsilon$, choisissez l'action qui maximise la valeur de Q pour l'état courant (exploitation).

Exécution de l'Action et Observation de la Récompense :

Exécutez l'action a et observez la récompense r et le nouvel état s.

Mise à Jour de la Table Q :

Mettez à jour la valeur de Q pour l'état-action précédent (s, a) en utilisant la règle de mise à jour suivante :

$Q(s, a) = \lambda \times (r + \gamma \times \max_{a'} Q(s', a')) + (1 - \lambda) \times Q(s, a)$

La formule décrit comment mettre à jour la valeur $Q(s, a)$, qui est l'estimation de la valeur de prendre une action a dans un état s, en utilisant la récompense obtenue r et la meilleure estimation de la valeur future maximale $max_{a'}Q(s', a')$ dans le nouvel état s'. Les composants de la formule sont les suivants :

$Q(s, a)$: La valeur actuelle de l'action a dans l'état s.

λ **(lambda) :** Le taux d'apprentissage. Ce coefficient détermine à quel point la nouvelle information (la récompense plus la récompense future estimée) affectera la valeur Q actuelle. Une valeur proche de 0 fait que l'apprentissage est très lent (la valeur actuelle change peu), tandis qu'une valeur proche de 1 fait que l'apprentissage est rapide (la valeur actuelle change beaucoup à chaque mise à jour).

r : La récompense obtenue après avoir pris l'action a dans l'état s.

γ **(gamma) :** L'ajustement du paramètre gamma se fait lors de **l'exploitation** dans les algorithmes d'apprentissage par renforcement.Le facteur de dépréciation. Il pondère l'importance des récompenses futures. Un γ proche de 0 signifie que l'agent ne valorise que les récompenses immédiates, tandis qu'un γ proche de 1 signifie que l'agent valorise davantage les récompenses futures.

$\max_{a'} Q(s', a')$: La meilleure valeur Q estimée pour le prochain état s', maximisée sur toutes les actions possibles a'. Cela représente l'estimation de la meilleure récompense future que l'agent peut obtenir à partir du prochain état.
Répétition des Étapes Boucle d'Apprentissage : Répétition des étapes 3 à 5 : Dans le contexte du Q-learning, une fois que vous avez établi la manière dont l'agent choisit une action, observe les conséquences, et met à jour ses estimations de valeurs Q, ces étapes sont répétées continuellement à chaque pas de chaque épisode. Un "pas" ici correspond à un cycle unique de choix d'action, observation de la réaction de l'environnement, et mise à jour de la table Q. L'ensemble de ces pas forme un "épisode", qui commence généralement dans un état initial et se termine quand un état terminal est atteint ou après un nombre prédéfini de pas.

Fin d'un Épisode :

Un épisode peut se terminer pour plusieurs raisons, comme atteindre un objectif, échouer de manière irréversible, ou après un certain nombre de pas pour éviter des épisodes excessivement longs. À la fin de chaque épisode, l'agent a potentiellement acquis de nouvelles connaissances sur les conséquences des actions dans divers états, ce qui affinera ses décisions futures.

Réduction de ϵ

Stratégie $\epsilon - greedy$:

La stratégie $\epsilon - greedy$ joue un rôle crucial dans l'équilibre entre exploration (choisir des actions aléatoires) et exploitation (choisir les meilleures actions selon la table Q actuelle).

Au début de l'apprentissage, lorsque l'agent ne possede que peu ou pas de connaissance sur l'environnement, il est important de favoriser l'exploration pour découvrir de nouvelles actions et apprendre des récompenses associees a ces actions. **Par consequent, un petit epsilon est choisi, ce qui signifie que l'agent a une faible probabilité de choisir une action de maniere aléatoire plutot que d'exploiter les connaissances acquises(**$1 - \epsilon$**.**

Au fur et à mesure que l'agent acquiert de l'experience et que sa connaissance de l'environnement s'ameliore, il devient de moins en moins bénéfique de prendre des actions aléatoires. À ce stade, il est plus efficace pour l'agent d'exploiter les connaissances qu'il a acquises pour maximiser les récompenses à long terme. Par conséquent, l'epsilon peut être progressivement augmenté au fil du temps, ce qui réduit la probabilité d'exploration aléatoire et favorise davantage l'exploitation des connaissances actuelles.

Cependant, il est important de noter que même lorsque l'epsilon est augmenté, il ne devrait pas nécessairement être fixé à zéro. En laissant une petite probabilité d'exploration aléatoire même après que l'agent ait acquis une connaissance approfondie de l'environnement, on évite le risque de rester coincé dans des comportements sous-optimaux et on laisse une certaine marge de ma-

nœuvre pour découvrir de nouvelles stratégies potentiellement plus efficaces. Cette approche permet à l'agent de maintenir un équilibre entre exploration et exploitation, même à des stades avancés de l'apprentissage.

Méthode de réduction :

ϵ peut être diminué progressivement, souvent en fonction du nombre d'épisodes ou de pas, afin d'augmenter la fréquence d'exploitation des connaissances acquises. Cette diminution peut être linéaire ou exponentielle, selon le comportement désiré.

CHAPITRE 2

Résolution du problème du Labyrinthe

2.1 Résolution du problème :Labyrinthe :

1. États :

Les états dans le labyrinthe sont définis par les coordonnées de chaque case accessible sur la grille. Chaque case du labyrinthe que le robot peut occuper (toutes les cases sauf celles qui sont des murs) représente un état unique dans l'espace d'état.

2. Actions :

Le robot peut se déplacer dans huit directions possibles : haut, bas, gauche, droite, ainsi que les quatre diagonales (haut-gauche, haut-droite, bas-gauche, bas-droite). Chaque action est représentée par un déplacement de la position actuelle du robot :

Haut : $(-1, 0)$

Bas : $(1, 0)$

Gauche : $(0, -1)$

Droite : $(0, 1)$

Haut-gauche : $(-1, -1)$

Haut-droite : $(-1, 1)$

Bas-gauche : $(1, -1)$

Bas-droite : $(1, 1)$

3. Récompenses :

Case sortie : +100 points.

Mur : -10 points (le robot ne peut pas traverser les murs, donc cette récompense peut être utilisée pour indiquer une tentative infructueuse de traverser un mur).

A**utres cases :** -1 point pour chaque déplacement, incitant le robot à trouver le chemin le plus court possible.

2.1.1 Implémentation de l'algorithme de Qlearning

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches

class Maze:
    def __init__(self, grid, start, goal, wall_penalty=-10,
       step_penalty=-1, goal_reward=100):

        self.grid = np.array(grid) #Represente le labyrinthe ou 1
           indique un mur et 0 un espace libre
        self.start = start
        self.goal = goal
        """start et goal : Definissent les coordonnees de depart et
           de l'objectif dans le labyrinthe"""
        self.wall_penalty = wall_penalty
        self.step_penalty = step_penalty
        self.goal_reward = goal_reward
        """wall_penalty, step_penalty, goal_reward : Definissent les
            recompenses (ou penalit s) pour heurter un mur, faire
           un pas, et atteindre l objectif, respectivement"""
        self.actions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Right,
           Down, Left, Up

    def step(self, state, action):
```

```
    """Prend un etat(l'etat initial c est (0,0) et une action,
       calcule le nouvel etat resultant de cette action.
    Gere les collisions avec les murs et verifie si l'objectif est
       atteint.
    Retourne le nouvel etat, la recompense/penalite correspondante,
       et un booleen indiquant si l'objectif est atteint."""
        next_state = tuple(np.array(state) + np.array(action))
        """Par exemple, si l' tat  est (0, 0) et l'action est (1, 0)
           , cela donnera (0+1, 0+0), ce qui  quivaut     (1, 0)."""
        if (next_state[0] < 0 or next_state[0] >= self.grid.shape[0]
            or
            next_state[1] < 0 or next_state[1] >= self.grid.shape[1]
                or
            self.grid[next_state] == 1):  # Wall check
            return state, self.wall_penalty, False
        elif next_state == self.goal:
            return next_state, self.goal_reward, True
        else:
            return next_state, self.step_penalty, False
    """si le nouvel etat resulte en une position en dehors des
       limites de la grille ou s'il rencontre un mur (represente par
        la valeur 1 dans la grille). Si l'une de ces conditions est
       vraie, cela signifie que l'agent a rencontre un mur ou a
       depasse les limites de la grille. Dans ce cas, la methode
       retourne l'etat actuel (state), une penalite pour avoir
       rencontr  un mur (self.wall_penalty), et un booleen
       indiquant que l'objectif n'a pas   t   atteint (False)."""
    def render(self, path):
    """Visualise le labyrinthe en dessinant les murs, le chemin
       parcouru, et marque les positions speciales (depart, objectif
       ) avec des couleurs distinctes.
```

```
35      Utilise matplotlib pour la visualisation, avec des carres pour
            les murs et des points pour le chemin."""
36          fig, ax = plt.subplots()
37          # Set up the grid for visualization
38          for y in range(self.grid.shape[0]):
39              for x in range(self.grid.shape[1]):
40                  if self.grid[y, x] == 1:  # Wall
41                      ax.add_patch(patches.Rectangle((x, y), 1, 1,
                                color='blue'))
42                  if (y, x) == self.start:  # Start
43                      ax.add_patch(patches.Rectangle((x, y), 1, 1,
                                color='red'))
44                  if (y, x) == self.goal:  # Goal
45                      ax.add_patch(patches.Rectangle((x, y), 1, 1,
                                color='green'))
46          # Draw path with small dots
47          for (y, x) in path:
48              ax.plot(x + 0.5, y + 0.5, 'yo', markersize=10, marker='o
                    ')  # Centered yellow dots
49
50          plt.xlim(0, self.grid.shape[1])
51          plt.ylim(0, self.grid.shape[0])
52          plt.gca().invert_yaxis()
53          plt.grid(True)
54          plt.xticks(np.arange(0, self.grid.shape[1], 1))
55          plt.yticks(np.arange(0, self.grid.shape[0], 1))
56          plt.show()
57      def render1(self, path, title, num_cycles):
58          fig, ax = plt.subplots()
59          for y in range(self.grid.shape[0]):
60              for x in range(self.grid.shape[1]):
```

```
61                  if self.grid[y, x] == 1:
62                      ax.add_patch(patches.Rectangle((x, y), 1, 1,
                            color='blue'))
63                  elif (y, x) == self.start:
64                      ax.add_patch(patches.Rectangle((x, y), 1, 1,
                            color='red'))
65                  elif (y, x) == self.goal:
66                      ax.add_patch(patches.Rectangle((x, y), 1, 1,
                            color='green'))
67          for (y, x) in path:
68              ax.plot(x + 0.5, y + 0.5, 'yo', markersize=10)
69          ax.set_xlim(0, self.grid.shape[1])
70          ax.set_ylim(0, self.grid.shape[0])
71          ax.set_title(f'{title} - Cycles: {num_cycles}')
72          plt.gca().invert_yaxis()
73          plt.grid(True)
74          plt.show()
75
76  def q_learning(maze, episodes, alpha=0.1, gamma=0.9, epsilon=0.1):
77  """Implemente l'algorithme Q-learning pour apprendre une politique
        optimale de navigation dans le labyrinthe.
78  episodes : Nombre total de fois que l'agent tentera de parcourir le
        labyrinthe pour apprendre.
79  alpha, gamma, epsilon : Parametres de l'algorithme (taux d'
        apprentissage, facteur de depreciation, et probabilite d'
        exploration).
80  Utilise une strategie epsilon-greedy pour equilibrer exploration et
        exploitation.
81  Met a jour la table Q en utilisant les recompenses obtenues et les
        estimations des meilleures recompenses futures."""
82      Q = np.zeros(maze.grid.shape + (4,))
```

```
83      for episode in range(episodes):
84          state = maze.start
85          done = False
86          #Extrait de Code Implementant epsilon-Greedy
87          while not done:
88              if np.random.random() < epsilon:
89                  action = np.random.choice(len(maze.actions))  #
                        Explore
90              else:
91                  action = np.argmax(Q[state])  # Exploit
92              next_state, reward, done = maze.step(state, maze.actions
                    [action])
93              old_value = Q[state][action]
94              future_value = np.max(Q[next_state])
95              Q[state][action] = old_value + alpha * (reward + gamma *
                    future_value - old_value)
96              state = next_state
97      return Q
98  """Le labyrinthe est initialise avec une grille specifiee, un point
       de depart, et un point d arrivee.
99  La fonction q_learning est appelee pour entrainer l agent avec un
       nombre defini d'episodes.
100 Apres l apprentissage, le chemin optimal est extrait en suivant la
       politique optimale derivee des valeurs Q apprises.
101 Le chemin optimal est visualise a l aide de la methode render."""
102 # Define the grid (0: free, 1: wall)
103 grid = [
104     [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
105     [0, 1, 0, 1, 1, 1, 1, 1, 1, 0],
106     [0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
107     [0, 1, 1, 1, 0, 1, 1, 1, 1, 0],
```

```
    [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
]
"""0 et 1 : Dans ce tableau, 0 represente un espace libre ou le
   robot peut se deplacer, et 1 represente un mur, o   le
   deplacement est bloque."""
start = (0, 0) #Coordonn es de la position de d part
goal = (4, 9) #Coordonn es de la position d'arriv e


# Initialize the maze
maze = Maze(grid, start, goal)

# Run Q-learning
Q = q_learning(maze, 1000) #nbr episode=1000

# Extract the optimal path from Q-values
state = start
path = [state]
while state != goal:
    action = np.argmax(Q[state])
    state = tuple(np.array(state) + np.array(maze.actions[action]))
    path.append(state)

# Visualize the path
maze.render(path)
```

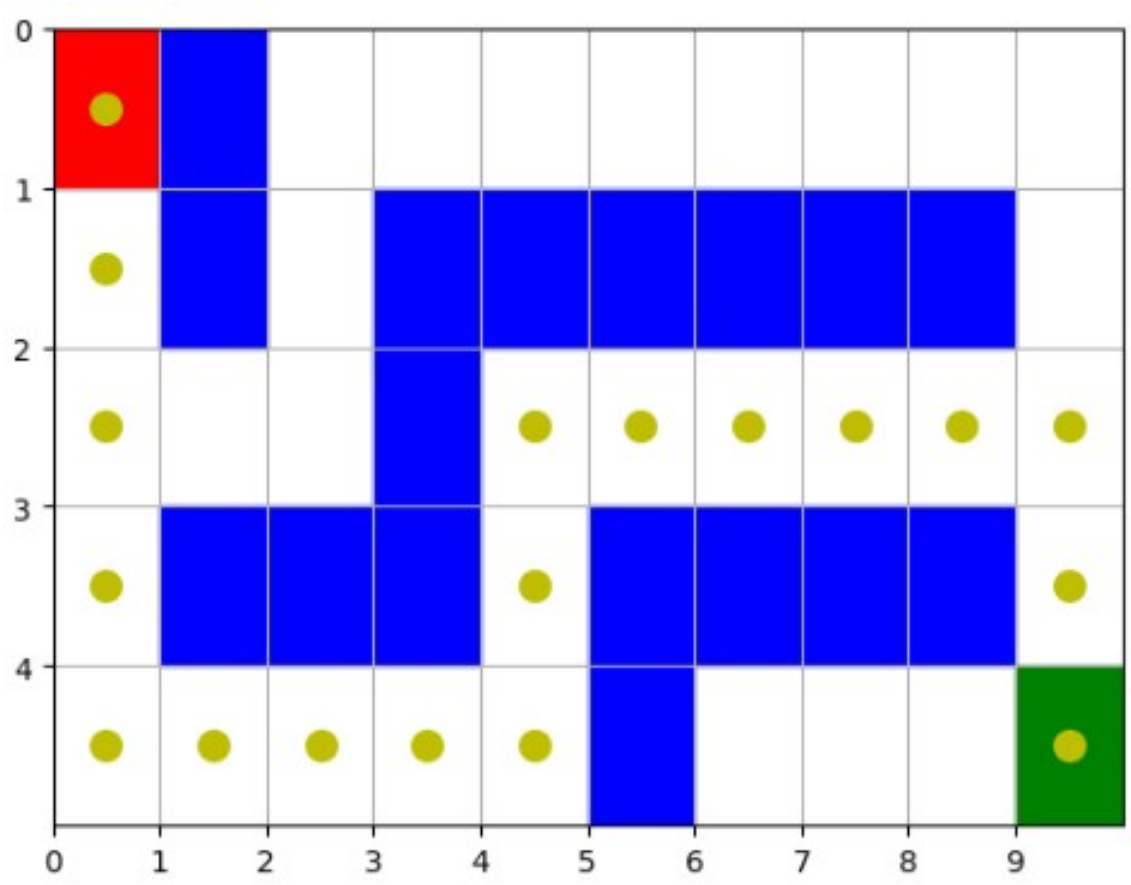

FIGURE2.1 – chemin du robot

Question3 : Faire varier ϵ, γ, (respectivement coefficient d apprentissage et facteur de depreciation) . donner à chaque fois le nombre de cycles pour trouver le chemin vers la sortie ainsi que le chemin trouvé.

Comparer les résultats obtenus à l'aide d'un graphique. Interpréter.

```
1 import numpy as np  # Necessaire pour les operations sur les
```

```
    tableaux

def q_learning(maze, episodes, alpha=0.1, gamma=0.9, epsilon=0.1):
    # Initialisation de la table Q avec des zeros pour toutes les
       paires etat action
    Q = np.zeros(maze.grid.shape + (4,))

    # Liste pour enregistrer le nombre de pas necessaires pour
       atteindre le but dans chaque episode
    steps_per_episode = []

    # Boucle sur le nombre total d'episodes specifies
    for episode in range(episodes):
        # Commence chaque episode a la position de depart du
           labyrinthe
        state = maze.start
        done = False  # Indicateur de fin d'episode, True si le but
           est atteint
        steps = 0  # Compteur de pas pour l'episode courant

        # Continue tant que l'objectif n'est pas atteint
        while not done:
            # Selection de l'action selon la politique epsilon-
               greedy
            if np.random.random() < epsilon:
                # Exploration: choix d'une action aleatoire
                action = np.random.choice(len(maze.actions))
            else:
                # Exploitation: choix de l'action avec la valeur Q
                   maximale pour l'etat actuel
                action = np.argmax(Q[state])
```

```
26
27              # Execution de l'action choisie et obtention du nouvel
                    etat et de la recompense
28              next_state, reward, done = maze.step(state, maze.actions
                    [action])
29
30              # Mise a jour de Q pour l'etat actuel et l'action selon
                    la formule du Q-learning
31              old_value = Q[state][action]
32              future_value = np.max(Q[next_state])
33              Q[state][action] = old_value + alpha * (reward + gamma *
                     future_value - old_value)
34
35              # Passage a l'etat suivant
36              state = next_state
37              steps += 1  # Incrementation du compteur de pas
38
39          # Enregistrement du nombre de pas dans la liste apres la fin
                de l'episode
40          steps_per_episode.append(steps)
41
42      # Retourne la table Q mise a jour et la liste du nombre de pas
            par episode
43      return Q, steps_per_episode
```

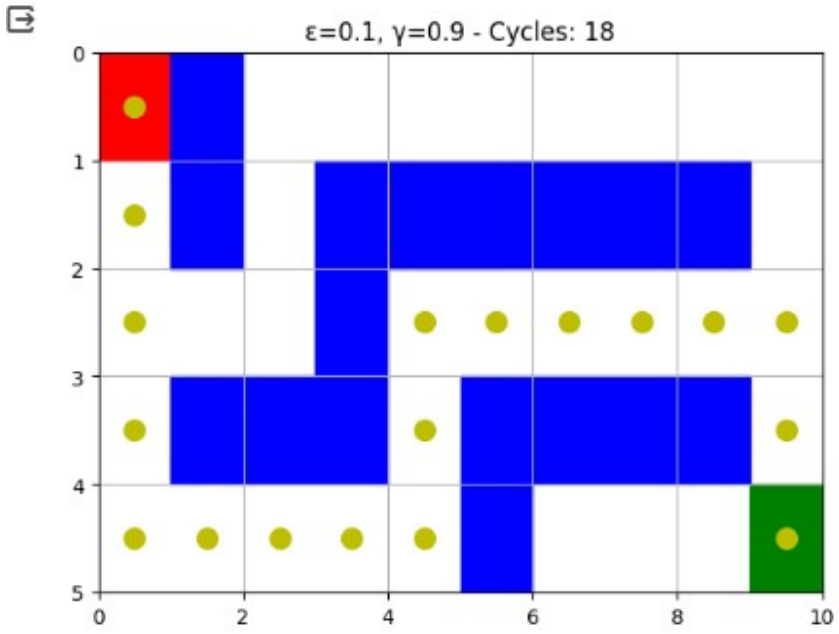

FIGURE2.2 – resultat1

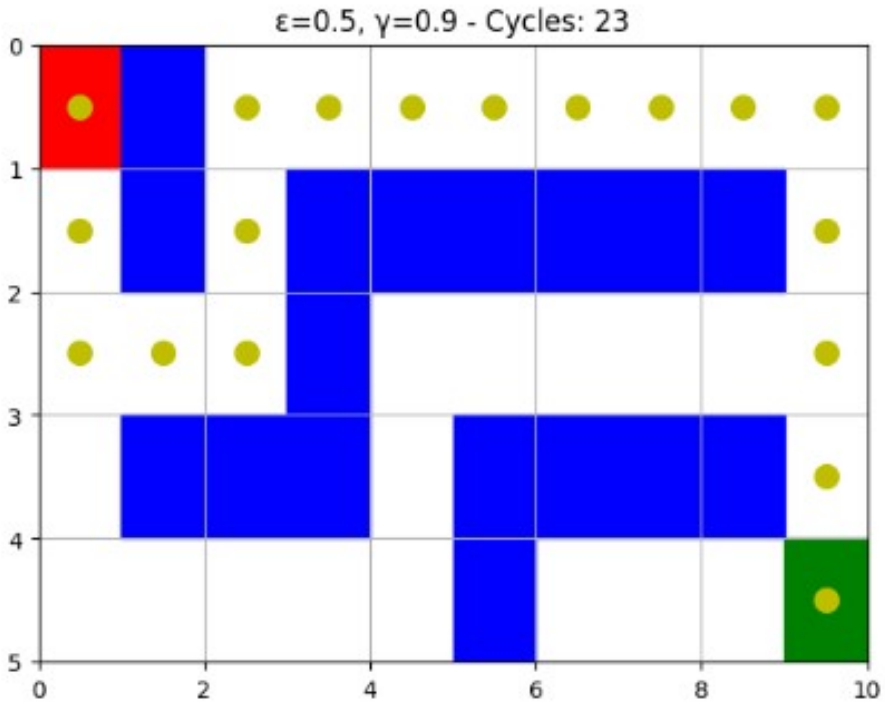

FIGURE2.3 – resultat2

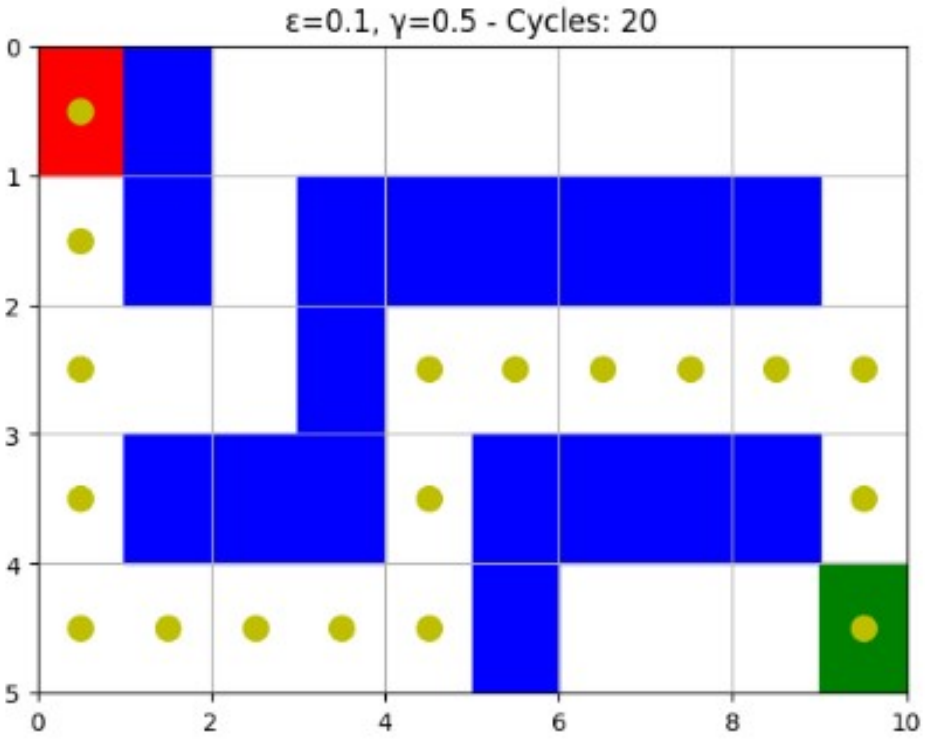

FIGURE2.4 – resultat3

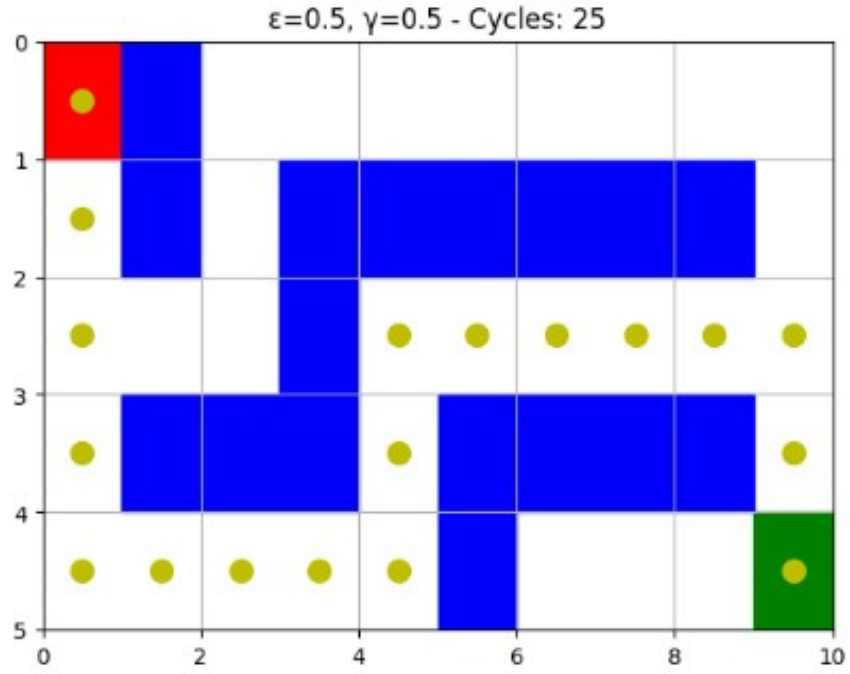

FIGURE2.5 – resultat4

2.1.2 Interpretation des Comparaison des resultats :

Première Image ($\epsilon = 0.1$, $\gamma = 0.9$, Cycles : 17) **Efficacité du chemin : Chemin le plus direct.**

Cycles pour l'achèvement : Moins de cycles parmi tous les scénarios.

Interprétation : Un epsilon plus faible avec un gamma élevé conduit à une exploration plus directe et efficace, mettant fortement l'accent sur les récompenses à long terme et moins sur l'exploration aléatoire.

Deuxième Image ($\epsilon = 0.5$, $\gamma = 0.9$, Cycles : 31) **Efficacité du chemin :** Plus d'exploration, chemin moins direct.

Cycles pour l'achèvement : Plus grand nombre de cycles.

Interprétation : Un epsilon plus élevé conduit à plus d'**exploration**, résultant en un chemin moins direct mais offrant plus d'apprentissage sur l'environnement. Ce scénario illustre le coût d'une exploration plus élevée avec une forte focalisation sur les récompenses futures.

Troisième Image ($\epsilon = 0.1$, $\gamma = 0.5$, Cycles : 19) **Efficacité du chemin :** Assez direct, légèrement plus de cycles que le plus faible epsilon et le plus haut gamma.

Cycles pour l'achèvement : Modérément efficace. **Interprétation :** Un epsilon plus faible avec un gamma plus bas conduit à un chemin plus direct mais légèrement moins efficace que le paramètre de gamma le plus élevé. Cela indique une approche équilibrée entre les récompenses immédiates et futures.

Quatrième Image (ϵ = 0.5, γ = 0.5 - Cycles : 30)

Efficacité du chemin : Exploration similaire à celle avec un gamma élevé mais moins efficace en termes de cycles par rapport à un gamma élevé.

Cycles pour l'achèvement : Élevé, similaire à une exploration élevée avec un gamma élevé. Interprétation : Une exploration accrue avec une focalisation équilibrée sur les récompenses immédiates et futures conduit à apprendre beaucoup sur l'environnement mais au prix de l'efficacité.

Analyse graphique et interprétation :

Pour fournir une interprétation plus claire, un graphique montrant les cycles pour chaque paramètre peut être tracé. Cependant, selon les images fournies :

*Un epsilon plus faible (0.1) conduit systématiquement à des chemins plus directs lorsque le gamma est élevé, ce qui indique qu'une exploration moins aléatoire combinée à une forte focalisation sur les récompenses futures (gamma élevé) est plus efficace pour cette configuration de labyrinthe particulière. Un epsilon plus élevé (0.5) augmente l'exploration, ce qui peut être bénéfique dans des labyrinthes plus complexes ou des environnements différents mais tend à augmenter le nombre de cycles nécessaires pour trouver le but dans ce scénario.

Conclusion :

Pour des labyrinthes similaires à ceux testés, un epsilon plus faible et un gamma plus élevé fournissent le parcours le plus efficace, suggérant que la minimisation des actions aléatoires tout en pondérant fortement les récompenses futures

est optimal. Cette stratégie, cependant, pourrait ne pas être aussi efficace dans des environnements où la connaissance initiale de l'environnement est faible, ou où l'environnement est très dynamique, car cela pourrait conduire à une convergence prématurée vers des chemins sous-optimaux. En revanche, une exploration plus élevée (epsilon plus élevé) peut être avantageuse dans des environnements plus stochastiques ou complexes où une exploration plus approfondie est nécessaire pour éviter les minima locaux.

Question 4 : Faire varier la taille du labyrinthe en choisissant aleatoirement des murs :

a. Grille 20x20

b. Grille 30x30

Comparer à l'aide d'un graphique, les resultats obtenus avec les 3 différentes tailles de labyrinthe.

```
def generate_random_maze(size, density=0.1):
    grid = np.zeros((size, size), dtype=int)  # Cree un tableau de
       zeros pour le labyrinthe
    start, goal = (0, 0), (size - 1, size - 1)  # Definit les
       positions de depart et d'arrivee
    obstacles = int(size * size * density)  # Calcule le nombre d'
       obstacles a placer

    # Place des obstacles sur le tableau
    while obstacles > 0:
        x, y = random.randint(0, size - 1), random.randint(0, size -
            1)
        if (x, y) != start and (x, y) != goal and grid[y, x] == 0:
            grid[y, x] = 1
            obstacles -= 1
    return grid, start, goal
"""Le but principal de la fonction simulate_maze est de tester et d'
   evaluer l'efficacite de l'algorithme Q-learning dans un
   environnement de labyrinthe, o  l'objectif est de naviguer d'un
   point de depart
a un point d'arrivee.
Cette fonction permet de mesurer combien de pas l'agent (le
   simulateur du labyrinthe)
```

```
16 prend pour atteindre l'objectif a travers differents episodes, en
      utilisant divers parametres de l'algorithme Q-learning."""
17 def simulate_maze(size, density=0.1, episodes=100, alpha=0.1, gamma
      =0.9, epsilon=0.1):
18     grid, start, goal = generate_random_maze(size, density)  #
          Genere le labyrinthe
19     steps_to_goal = []  # Stocke le nombre de pas pour atteindre l'
          objectif pour chaque episode
20 
21     Q = {}  # Initialisation du dictionnaire pour stocker les
          valeurs Q
22 
23     for episode in range(episodes):  # Boucle sur chaque episode
24         state = start  # Commence a la position de depart
25         steps = 0  # Compteur de pas
26 
27         while state != goal:  # Continue tant que l'objectif n'est
              pas atteint
28             if random.random() < epsilon:  # Decide aleatoirement d'
                  explorer
29                 action = random.choice([(0, 1), (1, 0), (0, -1),
                      (-1, 0)])
30             else:  # Exploite la meilleure action connue
31                 action = max([(0, 1), (1, 0), (0, -1), (-1, 0)], key
                      =lambda a: Q.get((state, a), 0))
32 
33             next_state = (state[0] + action[0], state[1] + action
                  [1])  # Calcule le prochain etat
34 
35             # Calcule la recompense et verifie si le prochain etat
                  est le but
```

```
36            if next_state == goal:
37                reward, done = 100, True
38            elif not (0 <= next_state[0] < size and 0 <= next_state
                  [1] < size) or grid[next_state] == 1:
39                reward, done = -10, False
40                next_state = state  # Revenir    l' tat  pr cedent
                      si c'est un mouvement invalide
41            else:
42                reward, done = -1, False
43
44            # Mise a jour de la valeur Q
45            q_key = (state, action)
46            old_value = Q.get(q_key, 0)
47            future_value = max([Q.get((next_state, a), 0) for a in
                  [(0, 1), (1, 0), (0, -1), (-1, 0)]])
48            Q[q_key] = old_value + alpha * (reward + gamma *
                  future_value - old_value)
49
50            state = next_state
51            steps += 1
52            if done:
53                break
54        steps_to_goal.append(steps)  # Ajoute le nombre de pas de
              cet episode a la liste
55
56    return steps_to_goal
```

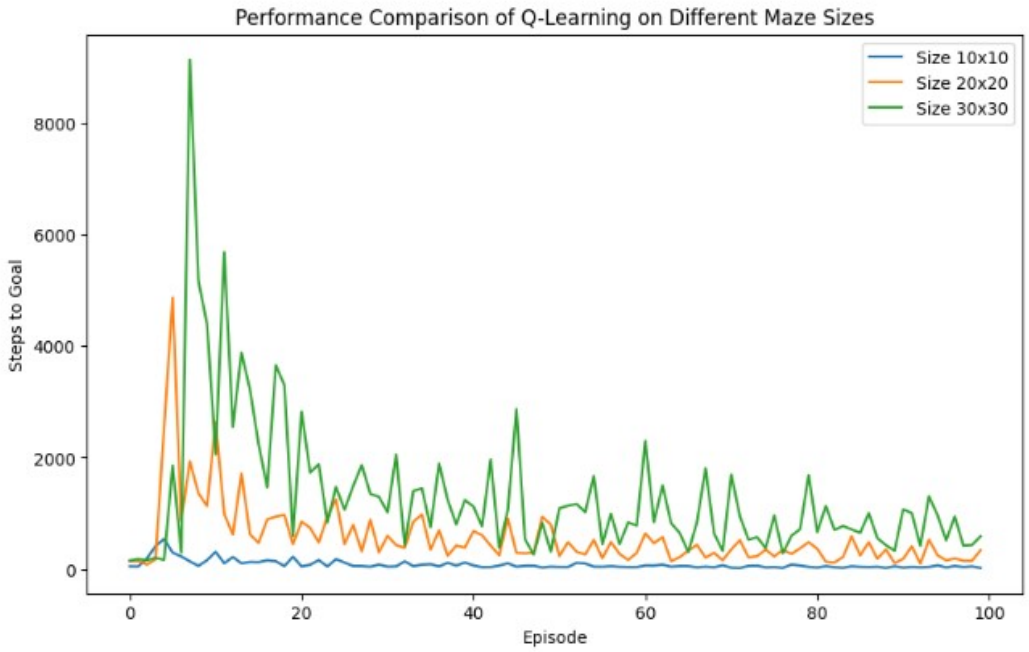

FIGURE2.6 – resultat

2.1.3 Analyse du Graphique

Décroissance du Nombre de Pas : On observe que pour toutes les tailles de labyrinthe, le nombre de pas nécessaires pour atteindre l'objectif diminue significativement au cours des premiers épisodes. Cela indique que l'algorithme apprend efficacement la structure du labyrinthe et améliore ses stratégies de navigation pour minimiser les pas.

2.1.3.1 Variabilité des Performances :

Labyrinthe 10x10 : Presente la plus rapide diminution du nombre de pas, avec une stabilisation rapide, ce qui suggere que l'algorithme parvient plus rapidement à optimiser son chemin dans un espace plus restreint.

Labyrinthe 20x20 : Montre une décroissance plus lente et plus de fluctuations dans le nombre de pas au fil des épisodes, ce qui peut refléter une complexité accrue et une difficulté plus grande pour l'algorithme à optimiser son approche dans un espace plus grand.

Labyrinthe 30x30 : Bien que les performances s'améliorent, les pics de variabilité restent élevés tout au long des simulations, indiquant des défis continus en termes d'optimisation de la navigation dans un espace encore plus vaste.

2.1.3.2 Comparaison des Tailles de Labyrinthe :

Les performances tendent à être moins stables dans les labyrinthes plus grands. Cela pourrait être dû à un plus grand nombre de configurations possibles de murs et de chemins, rendant l'apprentissage et la généralisation plus difficiles pour l'algorithme.

Conclusion et Implications Efficacité de l'Apprentissage : L'algorithme Q-learning montre une capacité notable à apprendre et à s'adapter à l'environnement du labyrinthe, comme en témoigne la réduction globale du nombre de pas nécessaires pour atteindre l'objectif au fil du temps.

Influence de la Taille du Labyrinthe : La taille du labyrinthe a un impact significatif sur la rapidité et la stabilité de l'apprentissage. Dans les environnements plus grands, il pourrait être nécessaire d'ajuster les paramètres de l'algorithme (comme epsilon, alpha, ou gamma) ou d'augmenter le nombre d'épisodes de formation pour obtenir des performances optimales.

Utilisation Pratique : Ces résultats peuvent guider la configuration des paramètres de l'algorithme lors de l'application de Q-learning à des problèmes réels de navigation ou de planification de trajet, en montrant l'importance de considérer la taille et la complexité de l'environnement lors de la conception et du réglage des algorithmes d'apprentissage par renforcement.

2.1.4 Visualisation du parcours du robot dans labyrinthe

Ce module définit une classe Maze pour simuler un labyrinthe où un agent peut naviguer en utilisant l'algorithme de Q-learning. La classe Maze est conçue pour générer dynamiquement un labyrinthe avec une densité spécifiée de murs et de positions aléatoires pour le départ et l'arrivée. Elle permet également de définir les récompenses et les pénalités pour différentes actions dans le labyrinthe.

Fonctionnalités : - Initialiser un labyrinthe avec des paramètres spécifiques pour la taille, la densité des murs, et les valeurs de récompenses/pénalités.

- Simuler le déplacement d'un agent dans le labyrinthe avec gestion des collisions et récompenses.

- Visualiser graphiquement le chemin parcouru par l'agent en temps réel, ce qui

aide à comprendre comment l'agent apprend et navigue dans le labyrinthe.

- Implémenter l'apprentissage par renforcement avec une stratégie d'action epsilon-greedy pour optimiser le chemin pris par l'agent en maximisant les récompenses cumulées.

- Extraire et afficher le chemin optimal à partir des valeurs Q apprises après une série d'épisodes d'apprentissage.

L'objectif est d'utiliser ce modèle pour étudier et démontrer les capacités de l'apprentissage par renforcement, en particulier l'algorithme de Q-learning, pour résoudre des problèmes de navigation complexes dans des environnements avec obstacles.

```
1 class Maze:
2     def __init__(self, grid_size, wall_density=0.2, wall_penalty
        =-10, step_penalty=-1, goal_reward=100):
3         # Initialisation du labyrinthe avec une taille specifiee,
             densite des murs, et des valeurs de penalite et de
             recompense.
4         self.grid = self.initialize_grid(grid_size, wall_density)  #
              Creation du grille avec des murs
5         self.start, self.goal = self.random_positions()  #
             Definition aleatoire des positions de depart et d'arrivee
6         self.actions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  #
             Definition des actions possibles : droite, bas, gauche,
             haut
7
8     def initialize_grid(self, grid_size, wall_density):
9         # Cree un grille ou les murs sont places aleatoirement selon
```

```
            la densite specifiee.
        return np.random.choice([0, 1], size=grid_size, p=[1-
            wall_density, wall_density])

    def step(self, state, action):
        # Determine le nouvel etat apres avoir pris une action, gere
            les collisions avec les murs et les recompenses.
        next_state = tuple(np.array(state) + np.array(action))
        if self.grid[next_state] == 1 or not (0 <= next_state[0] <
            self.grid_size[0] and 0 <= next_state[1] < self.grid_size
            [1]):
            return state, self.wall_penalty, False  # Reste en place
                si collision avec un mur, retourne une penalite
        elif next_state == self.goal:
            return next_state, self.goal_reward, True  # Retourne
                une grande recompense si l'objectif est atteint
        else:
            return next_state, self.step_penalty, False  # Mouvement
                normal avec un petit cout

    def render(self, path, interactive=False):
        # Affichage graphique du labyrinthe, des positions de depart
            et d'arrivee, et du chemin parcouru.
        for (y, x) in path:
            ax.plot(x + 0.5, y + 0.5, 'yo')  # Dessine le chemin en
                jaune
        if interactive:
            clear_output(wait=True)
            time.sleep(0.1)  # Attend un peu pour une visualisation
                en temps reel
        plt.show()
```

```
30  """Dans cette fonction, chaque fois que l'agent prend une action et
        met e jour son etat, la methode render est appelee pour
       visualiser l'etat actuel
31          du labyrinthe et le chemin parcouru par l'agent jusqu'e ce
               point. Cela permet de voir comment l'agent se deplace
               dans le labyrinthe, quels chemins il prend,
32          et comment il reagit aux recompenses et aux penalites."""
33  def real_time_q_learning(maze, episodes, alpha=0.1, gamma=0.9,
       epsilon=0.1):
34      # Implementation de l'apprentissage par renforcement Q-learning
           pour naviguer dans le labyrinthe.
35      while not done:
36          if np.random.random() < epsilon:
37              action = np.random.choice(len(maze.actions))  #
                   Exploration aleatoire
38          else:
39              action = np.argmax(Q[state])  # Exploitation : choisit
                   la meilleure action connue
40          next_state, reward, done = maze.step(state, maze.actions[
               action])
41          # Mise e jour de Q-value pour l'action prise
42          Q[state][action] = old_value + alpha * (reward + gamma *
               future_value - old_value)
43          path.append(state)  # Ajoute l'etat au chemin
44          maze.render(path, interactive=True)  # Affichage du chemin
               en temps reel
45
46  def extract_optimal_path(maze, Q):
47      # Extraction du chemin optimal base sur la table Q apres l'
           apprentissage.
48      while state != maze.goal:
```

```
49          action = np.argmax(Q[state])
50          state = tuple(np.array(state) + np.array(maze.actions[action
                ]))  # Avance vers le meilleur etat suivant
51          optimal_path.append(state)
```

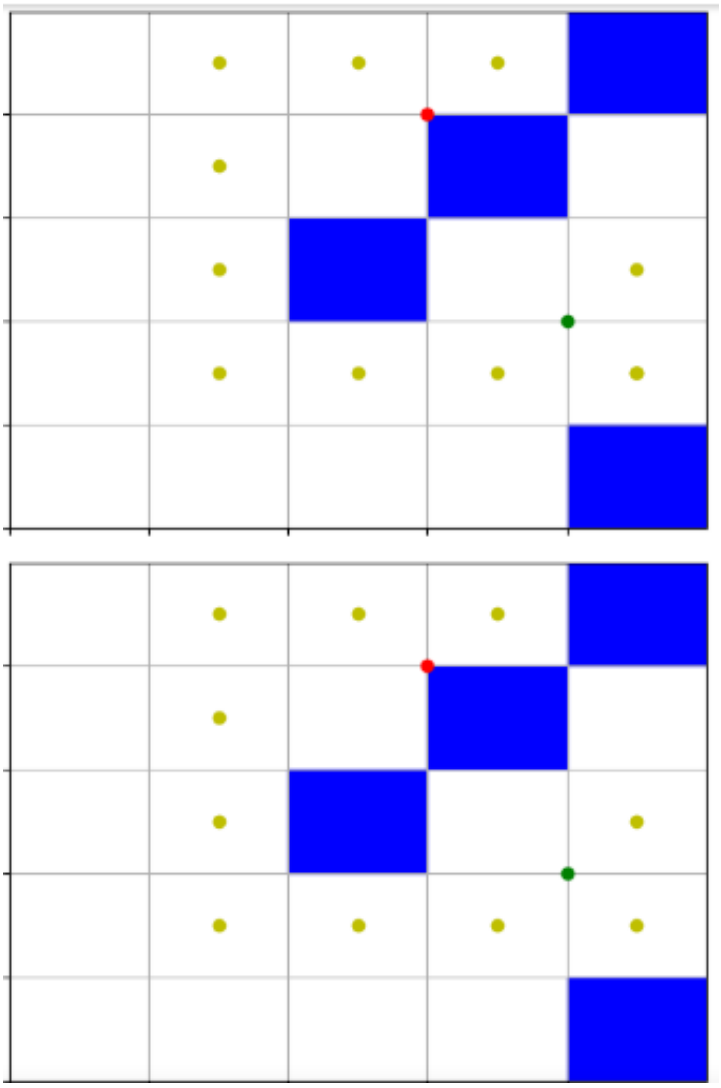

FIGURE2.7 – le output

=> L'algorithme de Q-learning démontre son efficacité à naviguer et trouver le chemin optimal dans le labyrinthe en s'adaptant intelligemment aux récompenses et pénalités rencontrées.