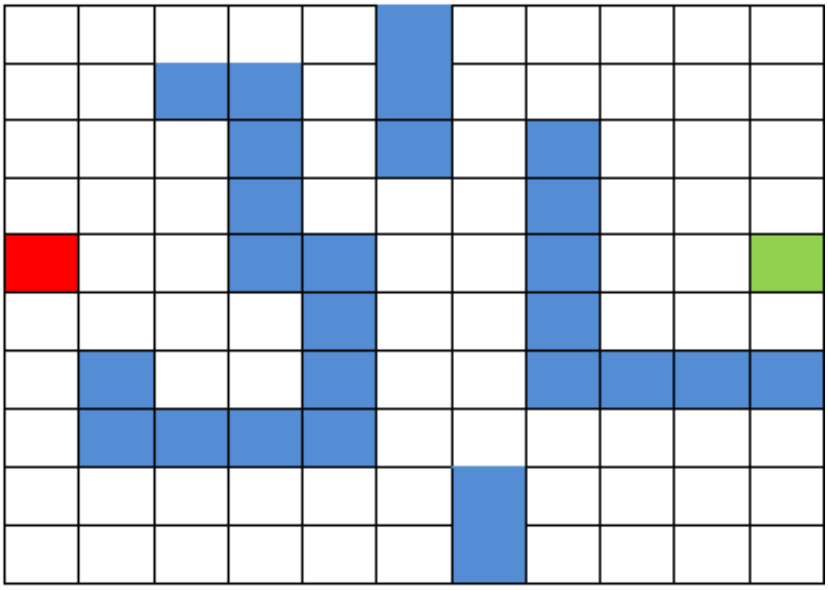
Problème du labyrinthe

Youssef Abdelhedi et Aicha Ettriki

Contents

1 Introduction

Le Q-learning est une technique d'apprentissage par renforcement dans le domaine de l'intelligence artificielle. Elle est utilisée pour résoudre des problèmes dans lesquelles un agent doit prendre des décisions séquentielles pour maximiser une récompense cumulée à long terme. Le Q-learning est un algorithme d'apprentissage par renforcement qui fonctionne en apprenant à estimer la valeur d'une action dans un état donné, représentée par une fonction appelée fonction Q. Cette fonction évalue la récompense attendue pour chaque action possible, permettant ainsi à l'agent d'agir de manière optimale dans son environnement. Le Q-learning est basé sur une méthode itérative où l'agent ajuste progressivement les estimations de la fonction Q en explorant et en exploitant différentes actions dans son environnement. Dans le cadre de ce rapport, nous explorerons le Q-learning, en mettant en prenant l'exemple du problème du labyrinthe. Ce problème met en avant un labyrinthe avec des passages, des murs et une sortie. Dans ce labyrinthe, un agent virtuel doit trouver son chemin vers la sortie en prenant une série de décisions à chaque pas. Chaque fois que l'agent atteint la sortie, il reçoit une récompense positive, tandis que s'il heurte un mur, il reçoit une récompense négative.



En utilisant le Q-learning, l'agent commence à explorer le labyrinthe de manière aléatoire, en prenant des décisions au hasard à chaque pas. À mesure qu'il explore, il met à jour ses estimations des valeurs Q, qui représentent la récompense attendue pour chaque action dans chaque état du labyrinthe. En apprenant par essais et erreurs, l'agent découvre progressivement les chemins les plus efficaces pour atteindre la sortie.

2 Pseudo-code du Q-learning

Voici une explication des étapes clés de cet algorithme :

1. **Initialisation :** Les valeurs $Q(s, a)$ sont initialisées arbitrairement pour toutes les paires état-action.
2. **Boucle principale :** Pour chaque épisode :
   (a) **Initialisation de l'état :** L'agent commence dans un état initial.
   (b) **Boucle de l'épisode :**
      i. **Choix de l'action :** L'agent choisit une action à prendre dans cet état.
      ii. **Exécution de l'action :** L'agent exécute l'action choisie dans l'environnement et observe la récompense résultante ainsi que l'état suivant.

iii. **Mise à jour de Q :** Les valeurs Q sont mises à jour en fonction de la récompense reçue, de la valeur de Q actuelle pour l'état-action, de la meilleure valeur Q pour l'état suivant, et des paramètres d'apprentissage α (taux d'apprentissage) et γ (facteur d'actualisation).

iv. **Transition d'état :** L'agent passe à l'état suivant et la boucle continue jusqu'à ce qu'un état terminal soit atteint.

(c) **Fin de l'épisode :** Une fois l'épisode terminé, l'algorithme revient à l'étape précédente et commence un nouvel épisode.

Algorithm 1 Q-Learning

Initialiser $Q(s, a)$ arbitrairement pour toutes les paires état-action
for chaque épisode **do**
    Initialiser l'état s
    while s n'est pas l'état terminal **do**
        Choisir une action a pour s en utilisant une politique dérivée de Q (par exemple, ϵ-greedy)
        Exécuter l'action a, observer la récompense r et l'état suivant s'
        Mettre à jour la valeur Q de l'état-action actuel :
            $Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$
        $s \leftarrow s'$
    end while
end for

3 Résolution du Problème du Labyrinthe avec le Q-learning

Problème du Labyrinthe et Q-learning :

Dans notre problème, l'agent doit naviguer à travers un labyrinthe pour atteindre une destination tout en évitant les obstacles. L'objectif est d'apprendre une politique de décision optimale qui guide l'agent vers la sortie du labyrinthe tout en minimisant le nombre de pas et en évitant les chemins sans issue.

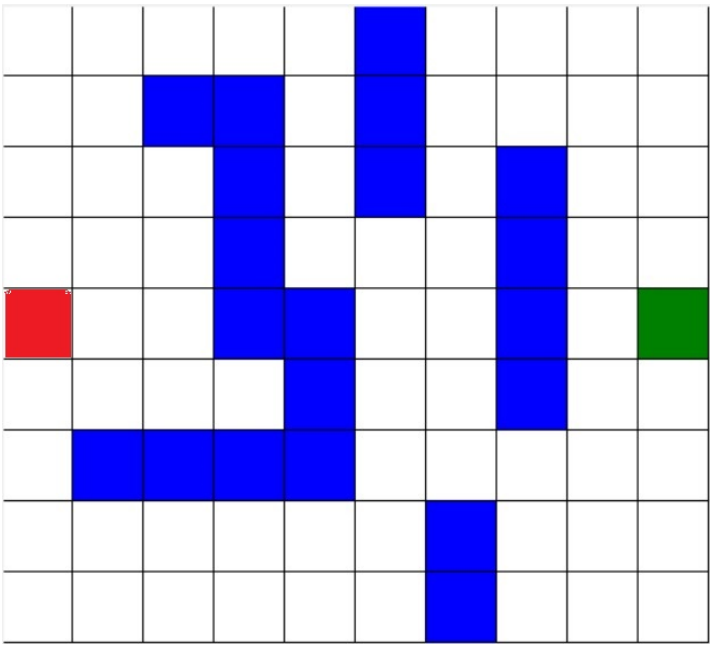
L'algorithme du Q-learning est une approche pour résoudre ce problème. Dans ce contexte, **les états représentent les positions possibles de l'agent dans le labyrinthe**, **les actions sont les mouvements disponibles** (par exemple, haut, bas, gauche, droite...), et **les récompenses sont définies pour chaque état-action**, avec une récompense positive pour atteindre la sortie et une récompense négative pour heurter un mur.

3.1 Mise en place de notre grille

Nous allons tout d'abord, définir une grille manuellement que nous allons utiliser comme support pour le labyrinthe.

```
# -1: case vide
# -10: mur infranchissable
# 100: case de sortie

grid=[[-1,-1,-1,-1,-1,-10,-1,-1,-1,-1],
      [-1,-1,-10,-10,-1,-10,-1,-1,-1,-1],
      [-1,-1,-1,-10,-1,-10,-1,-10,-1,-1],
      [-1,-1,-1,-10,-1,-1,-1,-10,-1,-1],
      [-1,-1,-1,-10,-10,-1,-1,-10,-1,100],
      [-1,-1,-1,-1,-10,-1,-1,-10,-1,-1],
      [-1,-10,-1,-1,-10,-1,-1,-10,-10,-10],
      [-1,-10,-10,-10,-10,-1,-1,-1,-1,-1],
      [-1,-1,-1,-1,-1,-1,-10,-1,-1,-1],
      [-1,-1,-1,-1,
```

Nous obtiendrons en sortie le graphique ci-dessous :



Les carrés bleus correspondent aux murs, le point rouge au point de départ et le point vert à la sortie. Nous créons ensuite une matrice Q qui est de forme $[état, action]$ et nous initialisons ses valeurs à 0. Par la suite, nous mettons à jour la matrice après chaque itération en suivant la formule suivante :

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_a Q(s',a) - Q(s,a) \right]$$

Ce tableau de valeurs devient une table de référence pour notre agent qui sélectionne la meilleure action en fonction des valeurs de cette matrice.

```
Q = np.zeros((len(grid), len(grid[0]), 8))
```

Ici, le troisième paramètre correspond à la dimension supplémentaire pour stocker les valeurs Q pour chaque paire (état, action). Dans ce contexte, le robot peut se déplacer de 8 façons différentes telles que haut, bas, gauche, droite mais aussi en diagonale.

3.2 Choix d'action (epsilon-greedy)

L'agent interagit avec son environnement de deux façons distinctes. Tout d'abord, il explore en agissant de manière aléatoire. C'est ce qu'on appelle l'exploration. Au lieu de sélectionner des actions basées sur la récompense future maximale, il choisit une action au hasard. Cette démarche est cruciale car elle permet à l'agent d'explorer et de découvrir de nouveaux états qui pourraient autrement être négligés lors du processus d'exploitation. Ensuite, l'agent se tourne vers l'exploitation en utilisant la table Q comme référence et en évaluant toutes les actions possibles pour un état donné. Il sélectionne alors l'action qui offre la plus grande valeur. Ce processus d'exploitation utilise les informations disponibles pour prendre des décisions.

Voici un code d'une fonction illustrant ce concept :

```
def choose_action(position, epsilon, Q):
    if random.random() < epsilon:
        return random.choice([0, 1, 2, 3,4,5,6,7])  # Exploration al atoire
    else:
        return np.argmax(Q[position[0], position[1]])  # Exploitation de la
            table Q
```

3.3 Déplacement du robot dans la grille

Nous allons créer la fonction `get_new_position(position, action,size)` qui représente une fonction de transition d'état pour le robot. Il se déplace dans la grille en fonction de l'action qu'il choisit à partir de son état actuel. Cette fonction prend en entrée l'état actuel du robot (ses coordonnées dans la grille) ainsi que l'action qu'il souhaite entreprendre. Celle-ci calcule ensuite les nouvelles coordonnées du robot en fonction de cette action, en vérifiant les limites de la grille.

```
    def get_new_position(position, action, size):
    i, j = position
    new_i, new_j = i, j
    if action == 0 and i > 0:
        new_i = i - 1  #haut
    elif action == 1 and i < size - 1:
        new_i = i + 1  # bas
    elif action == 2 and j > 0:
        new_j = j - 1  #gauche
    elif action == 3 and j < size - 1:
        new_j = j + 1  # droite
    elif action == 4 and i > 0 and j > 0:
        new_i, new_j = i - 1, j - 1  # haut gauche
    elif action == 5 and i > 0 and j < size - 1:
        new_i, new_j = i - 1, j + 1  #haut droite
    elif action == 6 and i < size - 1 and j > 0:
        new_i, new_j = i + 1, j - 1  # bas gauche
    elif action == 7 and i < size - 1 and j < size - 1:
        new_i, new_j = i + 1, j + 1  # bas droite
    if 0 <= new_i < size and 0 <= new_j < size: #limite  de la grille
        return (new_i, new_j)
    return position
```

3.4 Implémentation de l'algorithme Q-learning

Nous avons utilisé la bibliothèque Tkinter pour créer une interface graphique montrant le processus d'apprentissage de l'agent dans la grille. Cela nous a permis une meilleure compréhension du comportement de l'agent.

1. **Initialisation des variables** :
   - `Q` : La matrice Q
   - `robot_pos` : La position initiale de l'agent dans la grille.
2. **Boucle principale des itérations** : À chaque itération, la position du robot est réinitialisée à la position de départ `(4, 0)`.
3. **Boucle principale d'apprentissage** :
   - La boucle `while` continue jusqu'à ce que l'agent atteigne l'objectif (c'est-à-dire reçoive une récompense de 100).
   - À chaque étape, l'agent choisit une action en fonction de la politique epsilon-greedy (la fonction `choose_action` est appelée pour cela).
   - La nouvelle position de l'agent est calculée en fonction de l'action choisie.
   - La récompense pour cette action est obtenue à partir de la grille `grid` à la nouvelle position.
   - Ensuite, la meilleure récompense future pour la nouvelle position est calculée à partir de la matrice Q.
   - La valeur Q pour l'état et l'action actuels est mise à jour en utilisant la formule de mise à jour Q-learning.
   - La position actuelle est mise à jour avec la nouvelle position.

4. **Retour des valeurs apprises** :

   - Une fois toutes les itérations terminées, la matrice Q est renvoyée.

Listing 1: Algorithme Q-learning

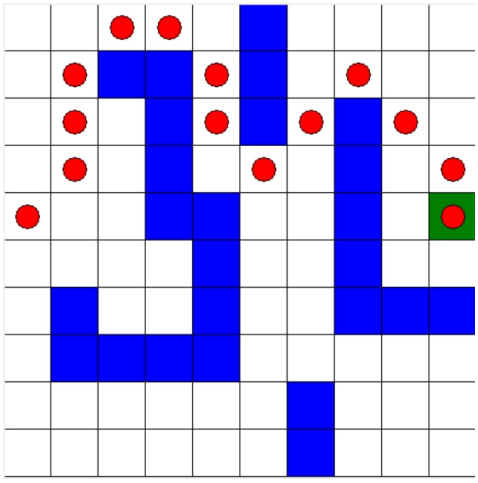
```
def Q_learning(canvas, epsilon, gamma, alpha, episodes, size,grid):
    Q = np.zeros((size, size, 8))
    robot_pos = (0, 0)
    for _ in tqdm(range(episodes)):
        position = (4, 0)
        reward=0
        while reward != 100:  # Tant que l'objectif n'est pas atteint
            action = choose_action(position, epsilon, Q)
            new_position = get_new_position(position, action, size)
            reward=grid[new_position[0]][new_position[1]]
            best_future_reward = np.max(Q[new_position[0], new_position[1]])
            Q[position[0], position[1], action] += alpha * (reward + gamma *
                best_future_reward - Q[position[0], position[1], action])
            position = new_position
            robot_pos = position
            plot_grid(canvas, grid, robot_pos)

    return Q
```

4 Chemin Optimal

Cette matrice Q nous permettra alors d'obtenir le chemin considéré comme meilleur par l'agent. A travers la fonction `find_optimal_path`, nous obtiendrons ce chemin en commençant à la position initiale (4,0) et en prenant au fur et à mesure la valeur maximale de la position de l'agent. A chaque fois que l'agent avance, la position est rajoutée au chemin.

```
def find_optimal_path(Q):
    size=grid.shape[0]
    optimal_path=[]
    pos=(4,0)
    while (pos not in optimal_path):
        optimal_path.append(pos)
        action=np.argmax(Q[pos[0], pos[1]])
        pos = get_new_state(pos, action, size)
    return optimal_path
```

Dans notre cas, cela nous a permis d'obtenir le chemin ci-dessous:



5 Variation des Paramètres ϵ et γ

5.1 Impact sur le Nombre de Cycles

Nous apportons une modification à notre fonction Q_learning afin de calculer le nombre de cycles ainsi que le chemin optimal.

```
    def Q_learning(epsilon, gamma, alpha, episodes, size, grid):
    Q = np.zeros((size, size, 8))
    robot_pos = (0, 0)
    episode_cycles = []
    paths = []
    for _ in (range(episodes)):
        position = (4, 0)
        reward = 0
        path = [position]
        cycles = 0
        while reward != 100:
            action = choose_action(position, epsilon, Q)
            new_position = get_new_position(position, action, size)
            reward = grid[new_position[0]][new_position[1]]
            best_future_reward = np.max(Q[new_position[0], new_position[1]])
            Q[position[0], position[1], action] += alpha * (reward + gamma *
                best_future_reward - Q[position[0], position[1], action])
            position = new_position
            path.append(position)
            cycles += 1
        episode_cycles.append(cycles)
        paths.append(path)
    return  episode_cycles, paths
```

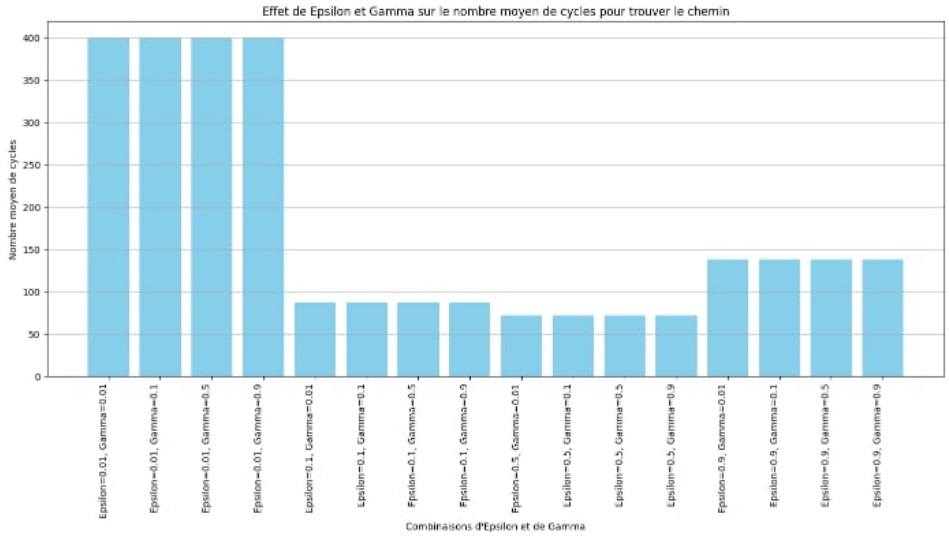
Ensuite, nous exécutons ce code qui explore différentes combinaisons de valeurs pour les hyper-paramètres ϵ et γ , en enregistrant le nombre de cycles et le nombre de pas trouvé pour le chemin optimal.

```
epsilon_values = [0.01, 0.1, 0.5, 0.9]
gamma_values = [0.01, 0.1, 0.5, 0.9]
results = {}
for epsilon in epsilon_values:
    for gamma in gamma_values:
            episode_cycles, paths = Q_learning(epsilon, gamma, 0.1, 1000, 10,
                grid)
            results[(epsilon, gamma)] = (episode_cycles, paths)
            print(f"epsilon={epsilon}, gamma={gamma}:")
            print(f"Nombre moyen de Cycle: {np.mean(episode_cycles)}")
            print(f"Nbr de pas trouv : {len(paths[np.argmin(episode_cycles)])
                 - 1}")
            print("-" * 20)
```

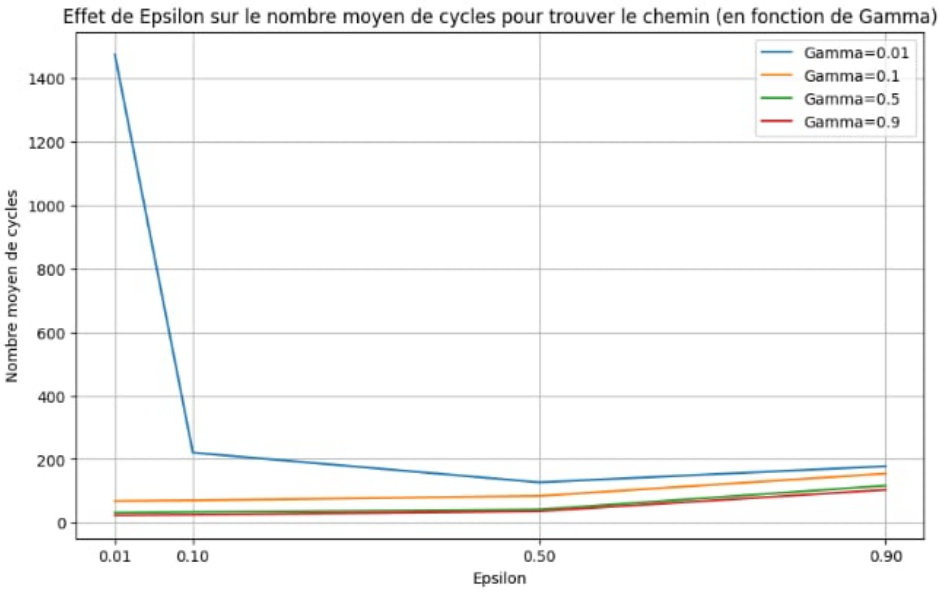
Voici, ci-dessous, le résultat du code :

```
epsilon=0.01, gamma=0.01:
Nombre moyen de Cycle: 1474.43
Nbr de pas trouvé: 29
--------------------
epsilon=0.01, gamma=0.1:
Nombre moyen de Cycle: 68.01
Nbr de pas trouvé: 11
--------------------
epsilon=0.01, gamma=0.5:
Nombre moyen de Cycle: 31.133
Nbr de pas trouvé: 12
--------------------
epsilon=0.01, gamma=0.9:
Nombre moyen de Cycle: 22.933
Nbr de pas trouvé: 11
--------------------
epsilon=0.1, gamma=0.01:
Nombre moyen de Cycle: 220.75
Nbr de pas trouvé: 10
--------------------
epsilon=0.1, gamma=0.1:
Nombre moyen de Cycle: 70.168
Nbr de pas trouvé: 10
--------------------
epsilon=0.1, gamma=0.5:
Nombre moyen de Cycle: 33.508
Nbr de pas trouvé: 11
--------------------
epsilon=0.1, gamma=0.9:
Nombre moyen de Cycle: 25.012
Nbr de pas trouvé: 10
--------------------
epsilon=0.5, gamma=0.01:
Nombre moyen de Cycle: 127.054
Nbr de pas trouvé: 12
--------------------
epsilon=0.5, gamma=0.1:
Nombre moyen de Cycle: 84.157
Nbr de pas trouvé: 10
--------------------
epsilon=0.5, gamma=0.5:
Nombre moyen de Cycle: 41.362
Nbr de pas trouvé: 9
--------------------
epsilon=0.5, gamma=0.9:
Nombre moyen de Cycle: 35.922
Nbr de pas trouvé: 10
--------------------
epsilon=0.9, gamma=0.01:
Nombre moyen de Cycle: 177.86
Nbr de pas trouvé: 13
--------------------
epsilon=0.9, gamma=0.1:
Nombre moyen de Cycle: 154.572
Nbr de pas trouvé: 10
--------------------
epsilon=0.9, gamma=0.5:
Nombre moyen de Cycle: 116.684
Nbr de pas trouvé: 14
--------------------
epsilon=0.9, gamma=0.9:
Nombre moyen de Cycle: 103.581
Nbr de pas trouvé: 11
--------------------
```

5.2 Quelques illustrations



Graphique à barres



5.3 Interprétation

1. Epsilon (ϵ) et Gamma (γ) faibles (0.01) : Avec ces valeurs, l'agent semble avoir besoin d'un grand nombre de cycles pour trouver le chemin, et le nombre moyen de pas trouvés est relativement élevé.

2. Epsilon (ϵ) et Gamma (γ) modéré à élevé : L'agent explore davantage son environnement (en raison d'un epsilon faible). Le nombre moyen de cycles pour trouver le chemin diminue considérablement par rapport aux valeurs de gamma plus faibles, et le nombre moyen de pas trouvés diminue également.

3. Epsilon (ϵ) et Gamma (γ) faible à modéré : Le nombre moyen de cycles pour trouver le chemin est généralement plus élevé que lorsque epsilon est faible, mais le nombre moyen de pas trouvés peut être réduit par rapport à des valeurs de gamma plus faibles.

4. Epsilon (ϵ) et Gamma (γ) élevés (0.9) : Un epsilon élevé favorise l'exploration plutôt que l'exploitation de la politique apprise. Avec ces valeurs, l'agent explore beaucoup plus son environnement, ce qui se traduit par un nombre plus élevé de cycles pour trouver le chemin mais souvent avec un nombre de pas trouvés plus faible.

Nous avons remarqué que pour faire le moins de cycles possible, les hyperparamètres à choisir parmis ceux sélectionnés sont epsilon=0.5 et gamma=0.01.

6 Création aléatoire

Nous avons créé une fonction qui initialise une grille aléatoirement en fonction d'une taille insérée.

```
1 def generate_grid(size):
2     grid = [[-10 if random.random() < 0.3 else -1 for _ in range(size)] for _
          in range(size)]
3     x = random.randint(0, size - 1)
4     y = random.randint(0, size - 1)
5     grid[x][y] = 100
6     return grid
```

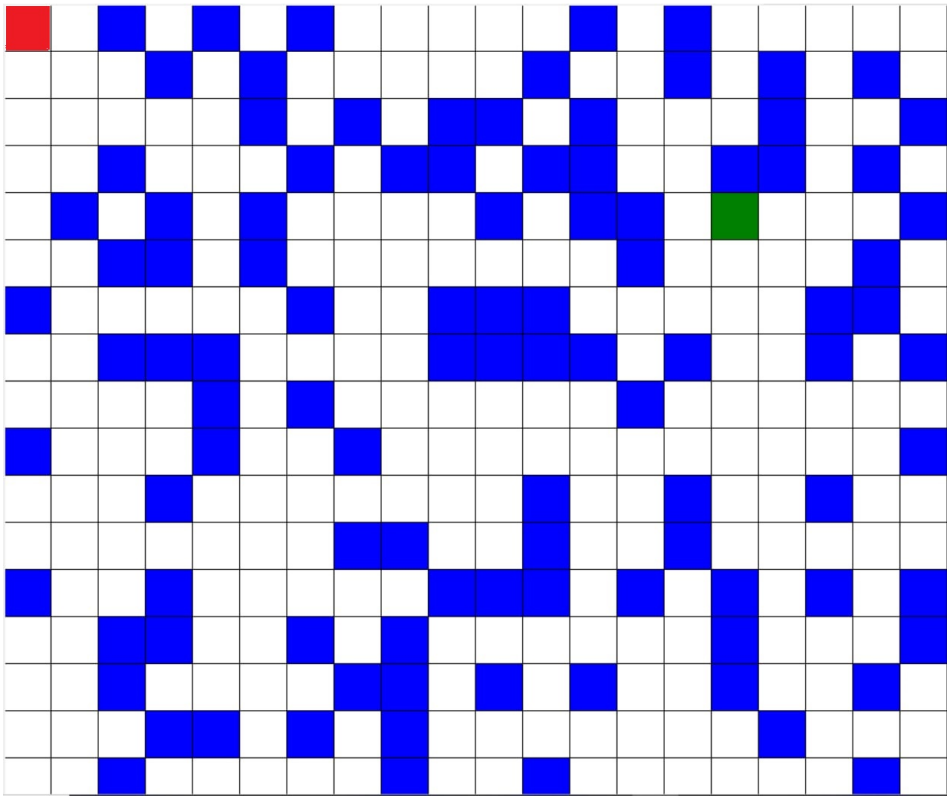
6.1 Grille 20x20

```
grid20 = generate_grid(20)
Matrice_Q20=Q_learning(canvas, 0.1, 0.9, alpha, 1000, 20,grid20)
```

Pour obtenir le chemin optimal, nous avons créé la fonction suivante :

```
def find_optimal_path(Q):
    size=grid.shape[0]
    optimal_path=[]
    pos=(0,0)
    while (pos not in optimal_path):
        optimal_path.append(pos)
        action=np.argmax(Q[pos[0], pos[1]])
        pos = get_new_position(pos, action, size)
    return optimal_path
path20=find_optimal_path(Matrice_Q20)
```

Nous aurons aléatoirement ce graph :



Grille 20x20

A ce moment, nous avons rencontré un problème. L'éxécution du code prenait énormément de temps. Pöur un exemple de grille de taille 20x20, l'algorithme de Q-learning a eu un temps estimé à 2 heures 30.

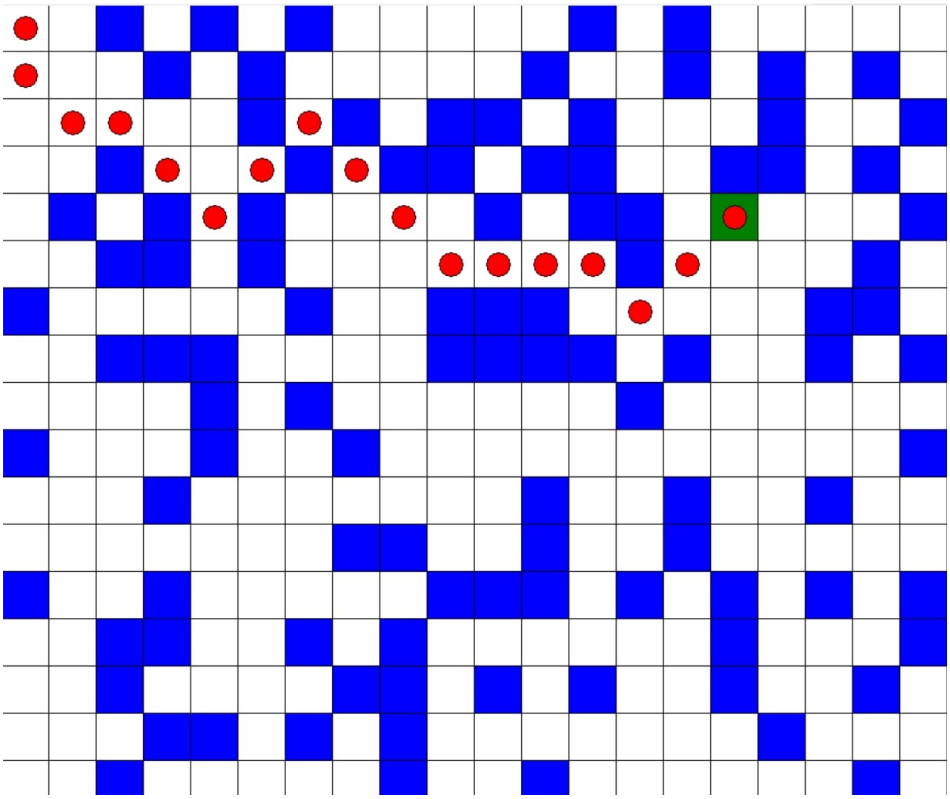
Nous avons alors décidé de récompenser davantage l'exploration de nouvelles cases plutôt que celles déja visitées. Les récompenses deviennent alors -10 pour les murs, -1 pour les cases visitées, 0 pour les cases non visitées et enfin 100 pour la case de sortie. Ainsi nous avons tout simplement changé le -1 du code qui génère automatiquement les grilles par un 0.

Et nous avons rajouté une condition de visite dans l'algorithme de Q-learning :

```
def Q_learning(canvas, epsilon, gamma, alpha, episodes, size,grid):
    Q = np.zeros((size, size, 8))
    robot_pos = (0, 0)
    for _ in tqdm(range(episodes)):
        state = (0, 0)
        reward=0
        grid2 = np.copy(grid)
        while reward != 100:  # Tant que l'objectif n'est pas atteint
            action = choose_action(state, epsilon, Q)
            new_state = get_new_state(state, action, size)
            reward=grid2[new_state[0]][new_state[1]]
            if reward==0:
                grid2[new_state[0]][new_state[1]]=-1
            best_future_reward = np.max(Q[new_state[0], new_state[1]])
            Q[state[0], state[1], action] += alpha * (reward + gamma *
                best_future_reward - Q[state[0], state[1], action])
            state = new_state
            robot_pos = state
            plot_grid(canvas, grid, robot_pos)
              # Pause pour visualiser l'apprentissage
    return Q
```

Nos efforts ont été effectivement récompensés étant donné que les milles itérations on été effectuées en 10:59 min pour le cas 20x20.

Voici donc le chemin obtenu par l'algorithme :

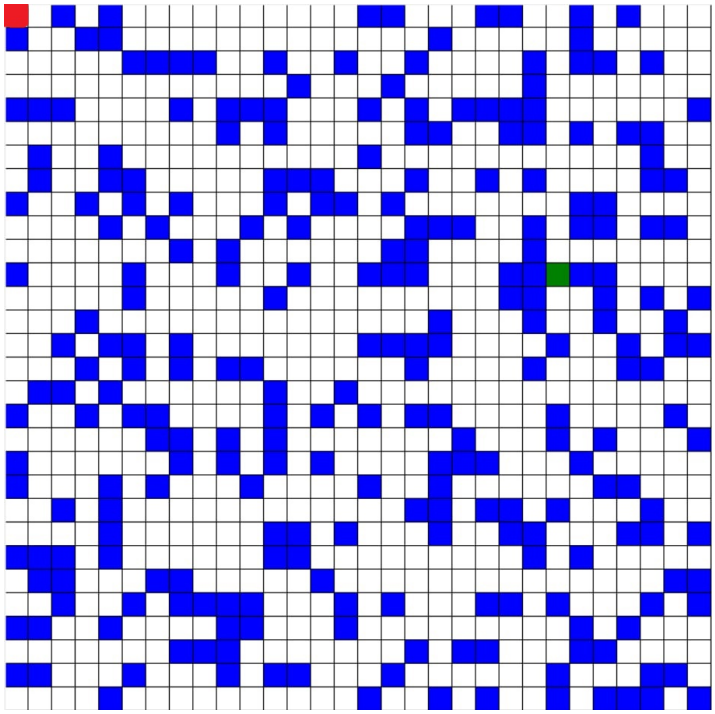


Chemin 20x20

6.2 Grille 30x30

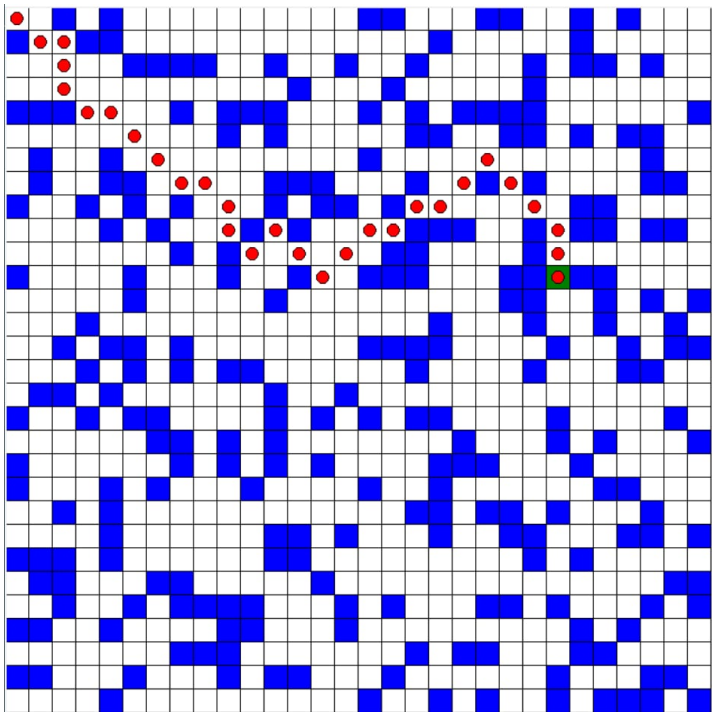
```
Matrice_Q30=Q_learning(canvas, 0.1, 0.9, alpha, 1000, 30,grid30)
path30=find_optimal_path(Matrice_Q30)
```

Nous avons obtenu en sortie ce graphique :



Grille 30x30

Avec les mêmes récompenses utilisées dans le cas de la grille 20x20, nous avons réussi à éxécuter l'algorithme en 30:54 min. Ainsi voici le résultat obtenu :



Chemin 30x30

7 Conclusion

- Grille 10x10 :
  Dans une grille 10x10 l'agent a un espace d'état relativement restreint avec seulement 100 cases, le Q-learning peut converger assez rapidement vers une politique optimale, car l'espace d'état est relativement petit. Les itérations nécessaires pour explorer et exploiter toutes les possibilités sont limitées, ce qui permet à l'agent de trouver rapidement une solution optimale.

- Grille 20x20 :
  Avec une grille deux fois plus grande, l'espace d'état augmente considérablement il est quatre fois plus grand que dans une grille 10x10, avec 400 cases. Le Q-learning nécessitera donc plus d'itérations pour explorer toutes les possibilités et converger vers une politique optimale. Cela signifie que l'agent pourrait prendre plus de temps pour trouver la meilleure solution, mais avec suffisamment d'itérations, il peut toujours y parvenir.

- Grille 30x30 :
  Dans une grille encore plus grande, comme celle de 30x30, l'espace d'état devient encore plus vaste. Le nombre d'itérations nécessaires pour que l'agent explore toutes les possibilités et converge vers une politique optimale augmentera de manière significative. Cela signifie que le temps nécessaire pour entraîner l'agent à prendre les meilleures décisions augmentera également. De plus, il peut être nécessaire d'optimiser l'algorithme ou d'utiliser des techniques avancées pour gérer l'explosion combinatoire des états possibles.

De plus, nous avons remarqué que nous devons comparer entre différents hyperparamètres tels que gamma, epsilon mais aussi les récompenses. En effet, différents hyperparamètres, le temps d'éxécution diffère énormément, mais également le nombre de cycles effectués ainsi que la taille du chemin optimal. Ces observations nous ont aidé à améliorer la qualité de nos algorithmes ainsi qu'à nous faire gagner du temps.