# Artificial Intelligence

## Computer Engineering Department

## Spring 2023

## Practical Assignment 3 - Reinforcement Learning

Mohammad Moshtaghi - Ali Salesi - Hossein Goli

## Personal Data

In [1]:
```
# Set your student number
student_number = '99101087'
first_name = 'AmirReza'
last_name = 'Azari'
```

In [2]:
```
!pip install gym[toy_text]

import gym
import numpy as np
from IPython.display import clear_output
import matplotlib.pyplot as plt
import os
from tqdm import trange
os.environ["SDL_VIDEODRIVER"] = "dummy"
clear_output() # You can use this method to clear your cell's output.
```

# Q1: Q-Learning (100 Points)

Author: Mohammad Moshtaghi
Please run all the cells.

# 1. Cliff Walking (70 pts)

In this section we are going to implement different Temporal Difference algorithms and compare their results. We start with a simple problem called **Cliff Walking**. You may have seen this game in your lecture slides and here we are going to train an RL Agent to play this game optimally.

First, lets get familiar with game's environment.

## 1-1. Environment (10 pts)

Lets declare our environment and see some of its hyperparameters.

In [25]:
```
env = gym.make('CliffWalking-v0', render_mode='rgb_array')
spec = gym.spec('CliffWalking-v0')

print(f"Action Space: {env.action_space}")
print(f"Observation Space: {env.observation_space}")
print(f"Max Episode Steps: {spec.max_episode_steps}")
print(f"Nondeterministic: {spec.nondeterministic}")
print(f"Reward Range: {env.reward_range}")
print(f"Reward Threshold: {spec.reward_threshold}\n")
```

```
Action Space: Discrete(4)
Observation Space: Discrete(48)
Max Episode Steps: None
Nondeterministic: False
Reward Range: (-inf, inf)
Reward Threshold: None
```

In [26]:
```
Actions =  {0: 'UP',
            1: 'RIGHT',
            2: 'DOWN',
            3: 'LEFT'}
```

You can use ***visualize*** function to draw your state.

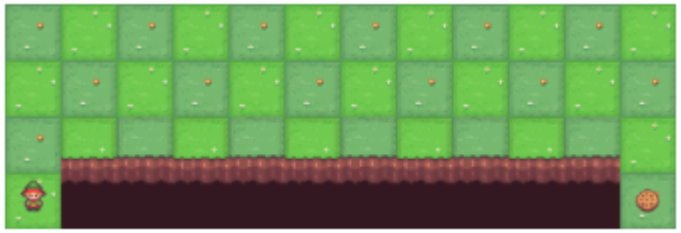
In [5]:
```
def visualize(env, action=None, reward=None):
    env_screen = env.render()
    plt.imshow(env_screen)
    plt.axis('off');
    title = ''
    if action:
        title += f'Action: {Actions[action]}'
    if reward:
        title += f'Reward: {reward}'

    plt.title(title)
    plt.show()
```

Test `visualize` function with a random action. First, using `env.reset` function, we reset our environment so that our agent returns to the starting point. for moving your agent, use `env.step` function. it returns four values:

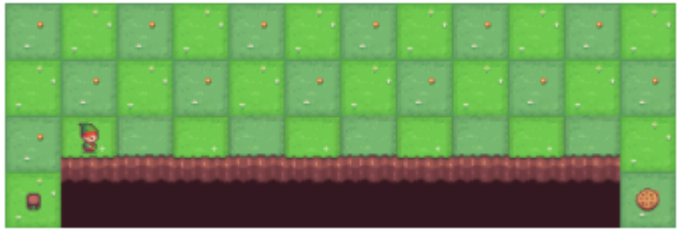
1. next_state
2. reward
3. done
4. some info (Honestly, it doesn't matter)

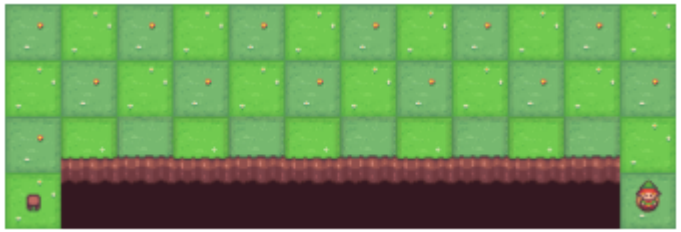
You may need this functions later :)

In [6]:
```
import random
env.reset() # You can use this function to reset your environment.
########################## YOUR CODE HERE ##########################
visualize(env)
env.step(0)
env.step(1)
visualize(env)
while True:
    rand = random.randint(0, 3)
    if env.step(rand)[2] == True:
        break
visualize(env)
########################## END YOUR CODE ###########################
```



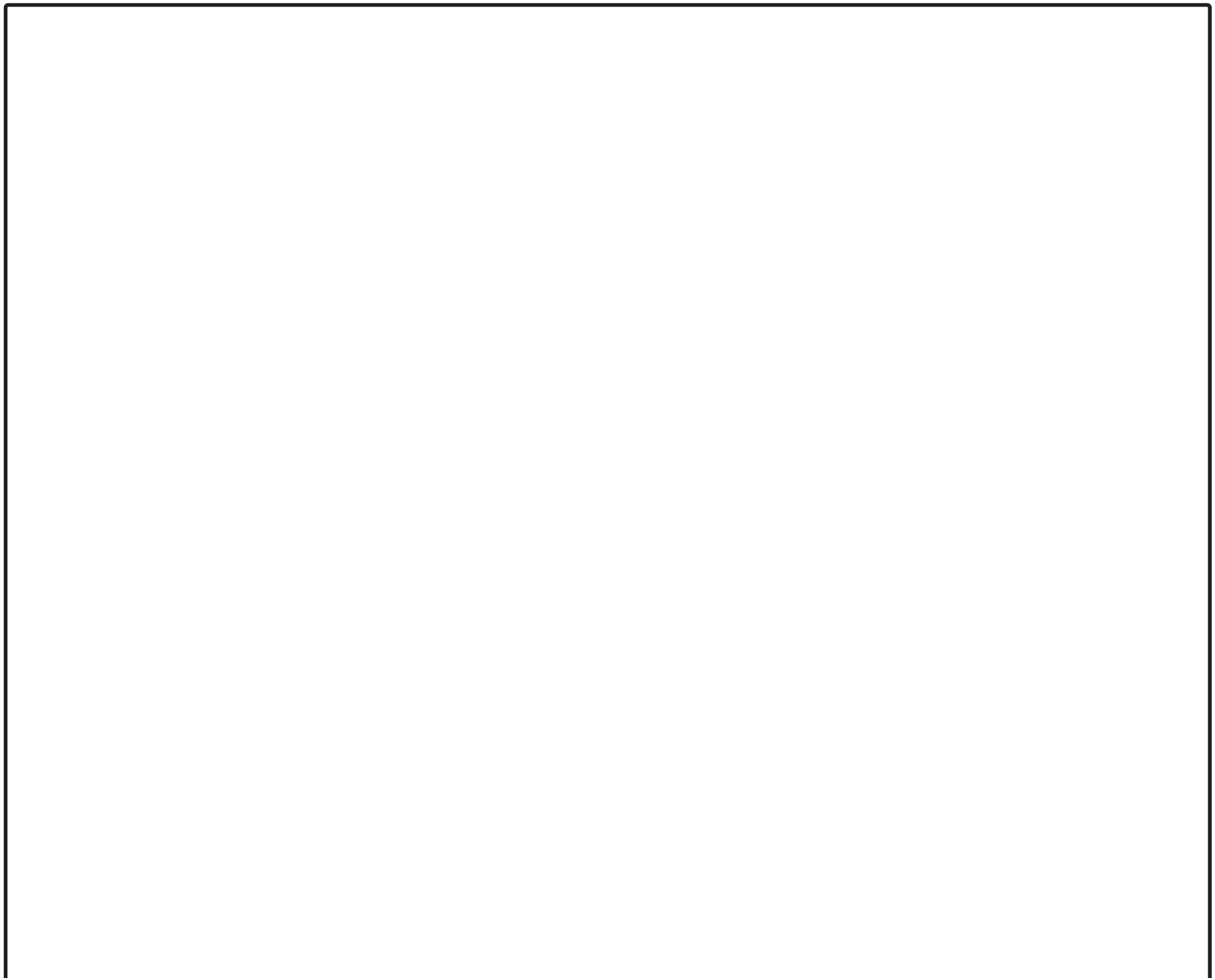
```
C:\Users\Amir Reza 81\anaconda3\lib\site-packages\gym\utils\passive_env_checker.py:233: Depr
ecationWarning: `np.bool8` is a deprecated alias for `np.bool_`.  (Deprecated NumPy 1.24)
  if not isinstance(terminated, (bool, np.bool8)):
```





## 1-2. Agent

Please read the class below. You must inherite this class in the following sections and implement different RL algorithms.



In [7]:
```
class Agent:

    def __init__(self, env, noise):
        self.q_values = []
        self.policy = {}            # I don't use this array
        self.env = env
        self.noise = noise
        ########################## YOUR CODE HERE ##########################
        # Declare any variables you need.
        ########################## END YOUR CODE ###########################

    def learn(self, num_episodes, alpha, gamma, epsilon):
        """
        Implement your Reinforcement Learning algorithm and train your agent in this function.
        At the end, you must fill the q_values array.

        Inputs:
            - alpha: Learning rate
            - gamma: Discount factor
            - epsilon: The probability that the agent will act randomly instead of greedy in sa
        """

        raise NotImplementedError()

    def create_policy(self, state, epsilon, number_of_actions):
        """                                                          **  I did not make this
                                                                     **  It is epsilon-greed
        Create your policy in this function after your agent learns the q_values.
        """
        ########################## YOUR CODE HERE ##########################
```

```
            rand = np.random.random()
            if (rand < epsilon):
                action = np.random.choice(number_of_actions)
            else:
                action = np.argmax(self.q_values[state])

            return action
            ######################### END YOUR CODE ##########################

        def act(self, action):
            """
            Move your agent one step according to your policy.
            """
            ######################### YOUR CODE HERE #########################
            state, reward, terminated = self.env.step(action)[0:3]
            return state, reward, terminated
            ######################### END YOUR CODE ##########################

        def evaluate(self, num_episodes):
            """
            Sample num_episodes episodes from your agent that acts according to your policy.
            Then return the average rewards it gets.

            Inputs:
                - num_episodes: Number of episodes for sampling.
            """
            ######################### YOUR CODE HERE #########################
            rewards = []

            for i in range(num_episodes):
                state = self.env.reset()[0]
```

```
        total_rewards = 0
        terminated = False
        #while not terminated:
        for j in range(200):
            if terminated:
                break
            rand = np.random.random()
            if self.noise > rand:
                action = np.random.choice(env.action_space.n)
            else:
                action = np.argmax(self.q_values[state][:])
            new_state, reward, terminated = self.act(action)
            total_rewards += reward
            state = new_state
            if i == num_episodes - 1:           # for displaying the last episodes
                visualize(env)
        rewards.append(total_rewards)
    return np.mean(rewards)
    ######################### END YOUR CODE ##########################
```

## 1-3. Q-Learning (15 pts)

In this section, you must use ***Q-Learning*** algorithm to train your agent. Note that the action-value function Q(s,a) is updated iteratively as follows:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left( r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right)$$

where `r` is the reward received after taking action a in state `s`, `s'` is the new state, $\gamma$ is the discount

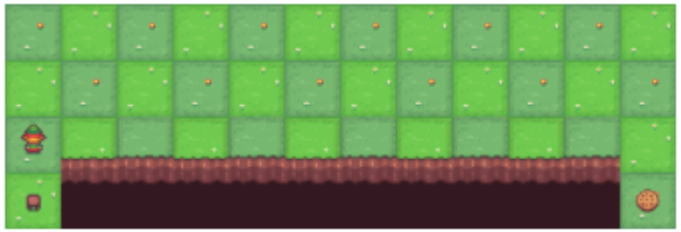
In [8]:
```
class CliffWalkerQL(Agent):

    def learn(self, num_episodes, alpha, gamma, epsilon):
        ######################### YOUR CODE HERE ##########################
        self.q_values = np.zeros((48, 4))
        for i in range(num_episodes):
            state = self.env.reset()[0]
            terminated = False
            while not terminated:
                action = self.create_policy(state, epsilon, 4)
                new_state, reward, terminated = self.act(action)
                self.q_values[state][action] = self.q_values[state][action] +\
                                               alpha * (reward + gamma * np.max(self.q_values[r
                                                        self.q_values[state][action])
                state = new_state
        ######################### END YOUR CODE ###########################
```

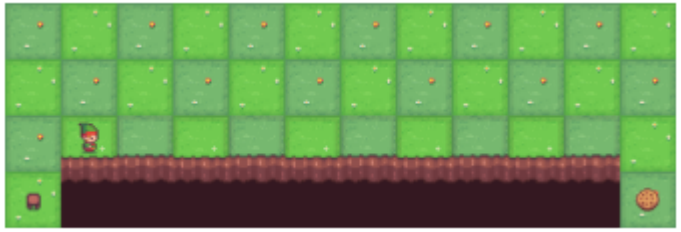
## 1-4. Q-Learning Evaluation (10 pts)

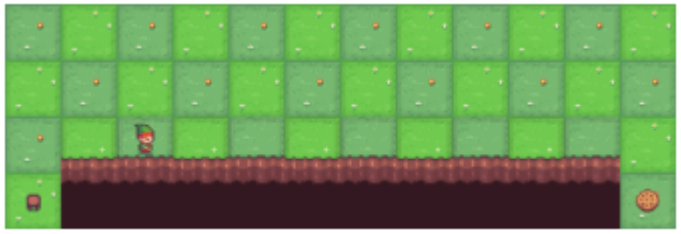
Train your agent and then evaluate it and display the result. Using the `visualize` function, show the path your agent takes in one of the episodes.

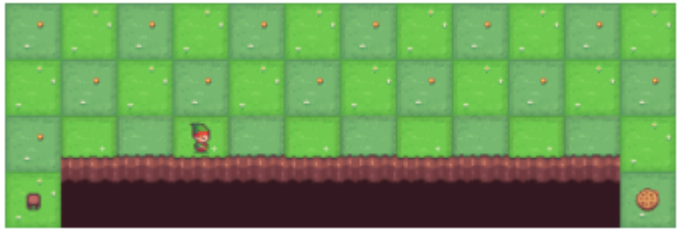
In [9]:
```
cliff_walker_ql = CliffWalkerQL(env, 0)
alpha = 0.8
gamma = 0.95
epsilon = 0.2
episodes = 1000
cliff_walker_ql.learn(episodes, alpha, gamma, epsilon)
```

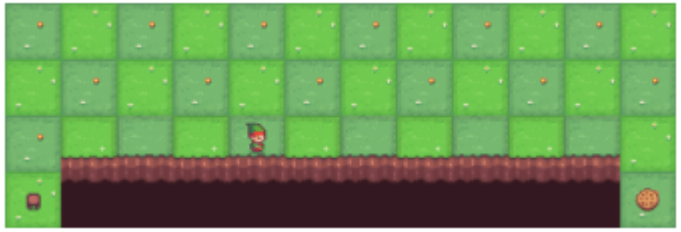
In [10]:
```
########################## YOUR CODE HERE ##########################
cliff_walker_ql.evaluate(episodes)
########################### END YOUR CODE ###########################
```

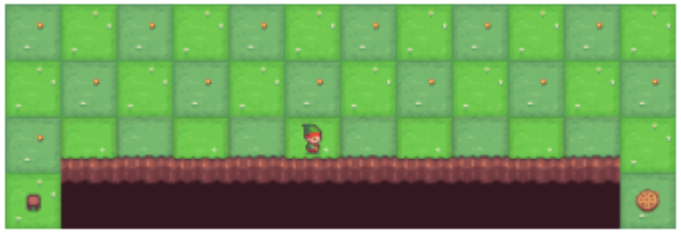


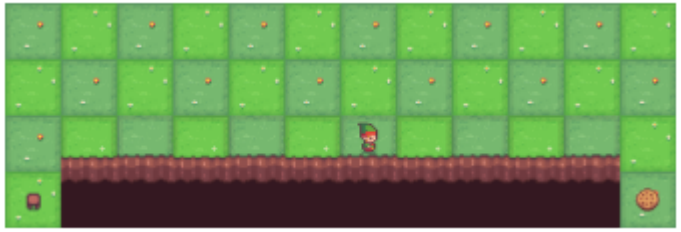


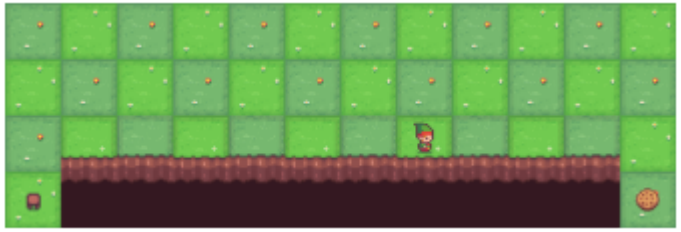


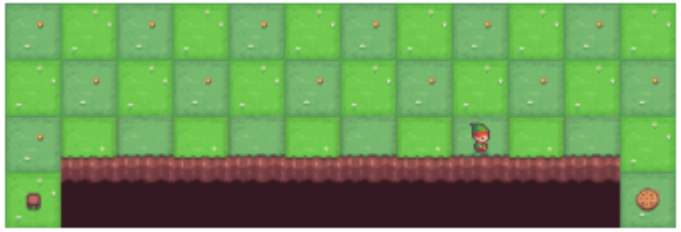


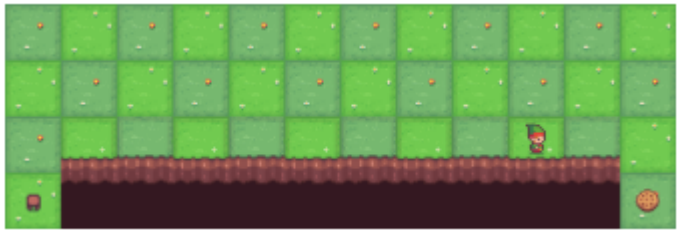


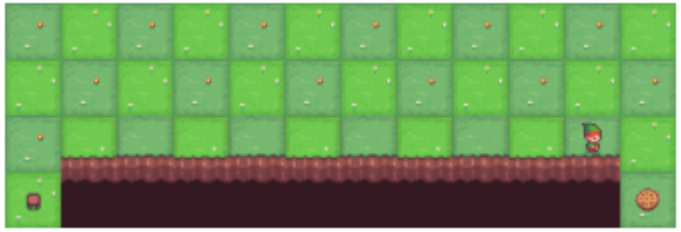


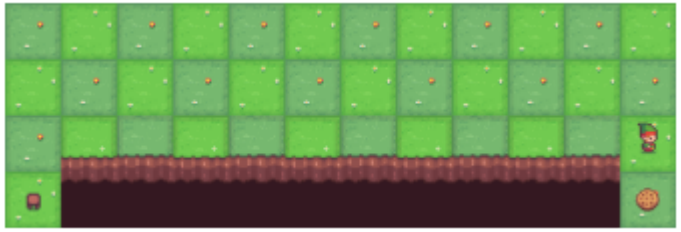


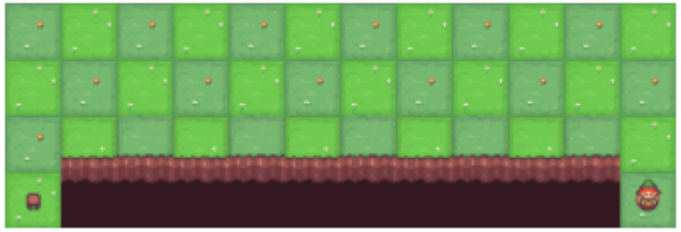












```
-13.0
```

## 1-5. SARSA (15 pts)

This time, you should use the SARSA algorithm, which is slightly different from the Q-Learning in implementation. But the result may significantly differ, and you should explain this difference, if any.

Note that the SARSA update rule can be represented as:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right)$$

Where $Q(S_t, A_t)$ is the current estimate of the expected return for taking action $A_t$ in state $S_t$, $\alpha$ is the learning rate, $R_{t+1}$ is the reward received after taking action $A_t$ in state $S_t$, $\gamma$ is the discount factor, and $Q(S_{t+1}, A_{t+1})$ is the estimated return for taking action $A_{t+1}$ in the next state $S_{t+1}$.

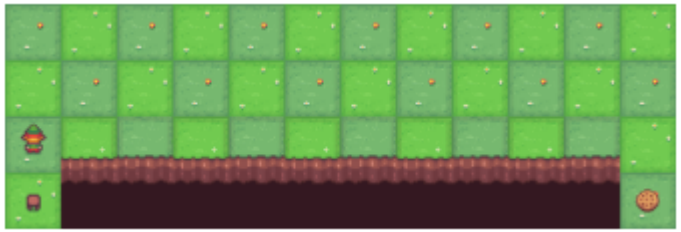
In [11]:
```
class CliffWalkerSARSA(Agent):

    def learn(self, num_episodes, alpha, gamma, epsilon):
        ########################## YOUR CODE HERE ##########################
        self.q_values = np.zeros((48, 4))
        for i in range(num_episodes):
            state = self.env.reset()[0]
            terminated = False
            action = self.create_policy(state, epsilon, 4)
            while not terminated:
                new_state, reward, terminated = self.act(action)
                next_action = self.create_policy(new_state, epsilon, 4)
                self.q_values[state][action] = self.q_values[state][action] +\
                                               alpha * (reward + gamma * (self.q_values[new_sta
                                                        self.q_values[state][action])

                state = new_state
                action = next_action
        ########################## END YOUR CODE ###########################
```

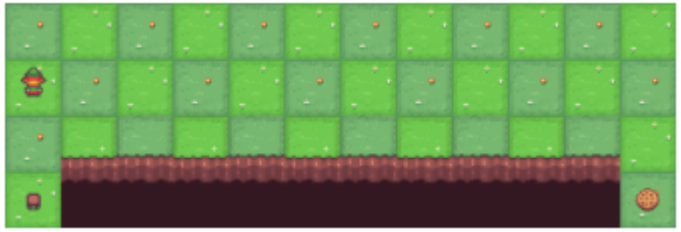
## 1-6. SARSA Evaluation (10 pts)

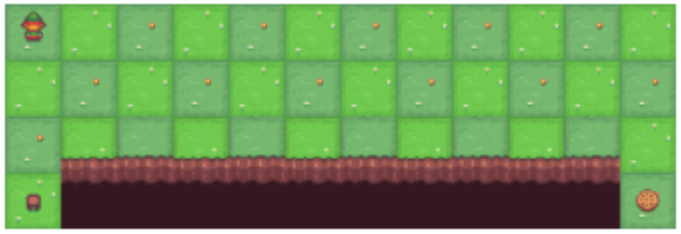
Train your agent and then evaluate it and display the result. Using the `visualize` function, show the path your agent takes in one of the episodes.

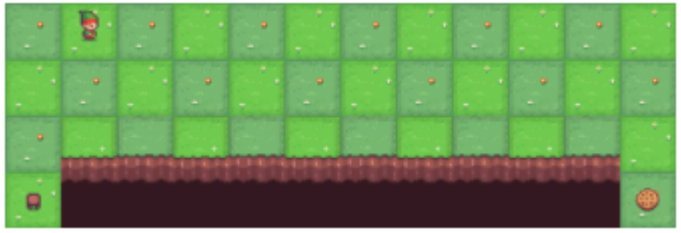
In [12]:
```
cliff_walker_sarsa = CliffWalkerSARSA(env, 0)
alpha = 0.1
gamma = 0.9
epsilon = 0.2
episodes = 1000
cliff_walker_sarsa.learn(episodes, alpha, gamma, epsilon)
```

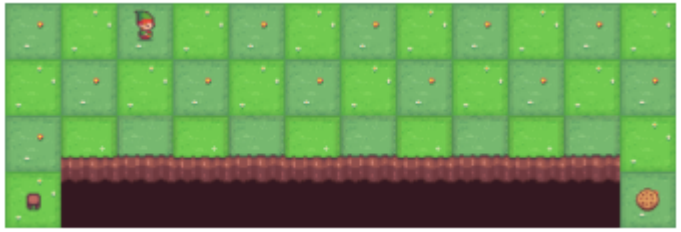
In [13]:
```
######################### YOUR CODE HERE ##########################
cliff_walker_sarsa.evaluate(episodes)
########################## END YOUR CODE ##########################
```

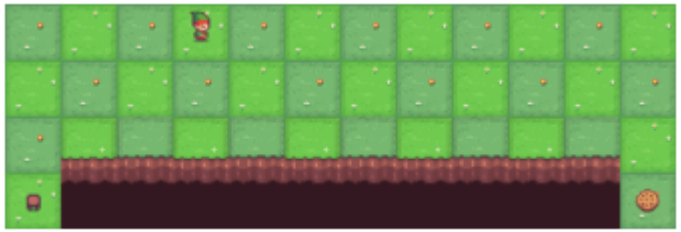


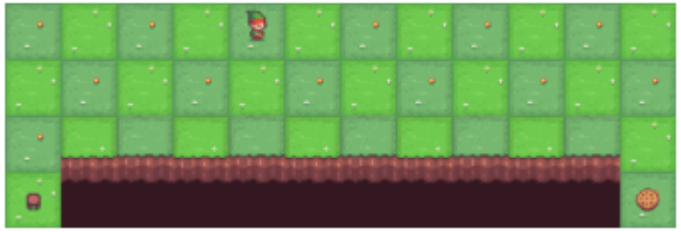


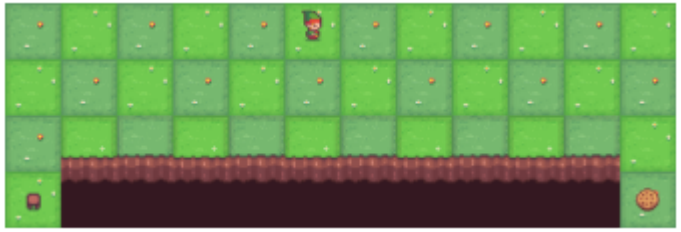


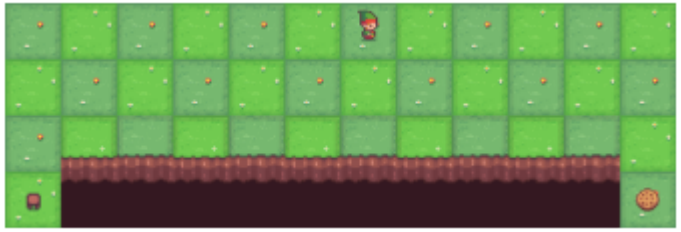


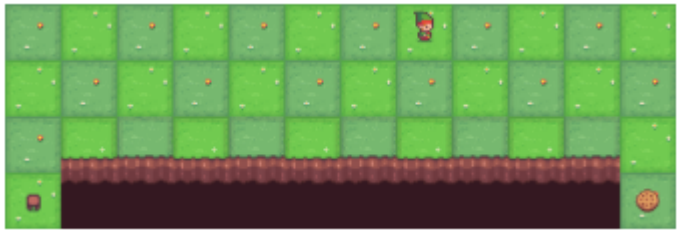


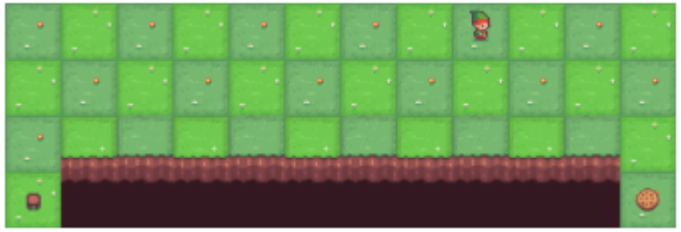


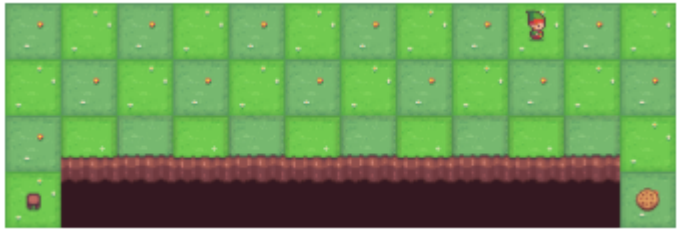


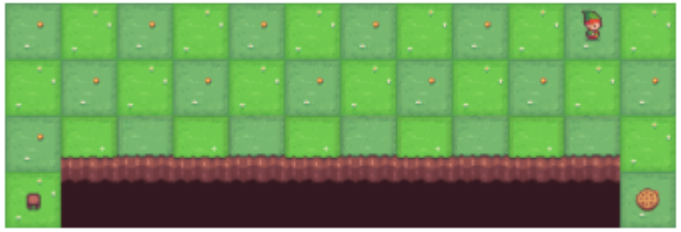


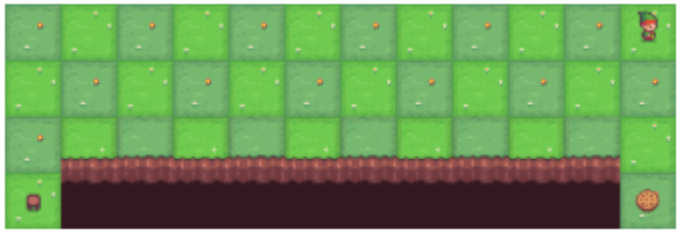


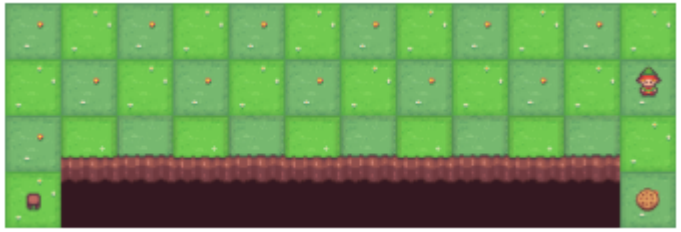


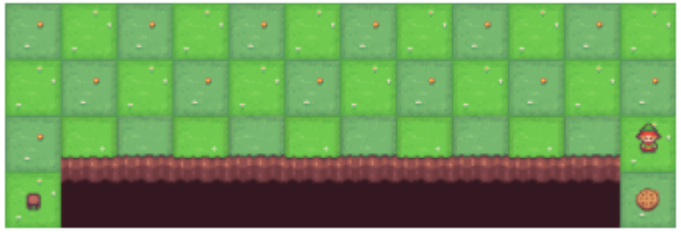


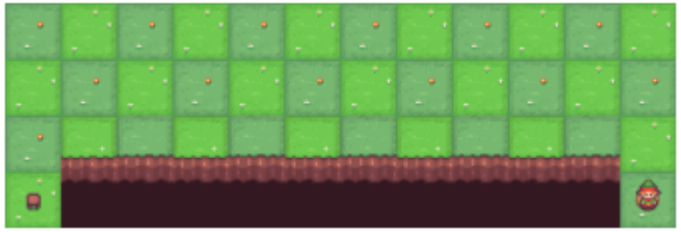










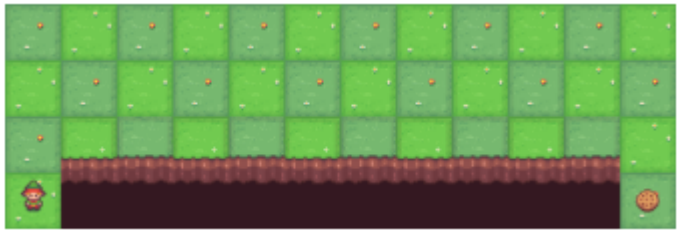


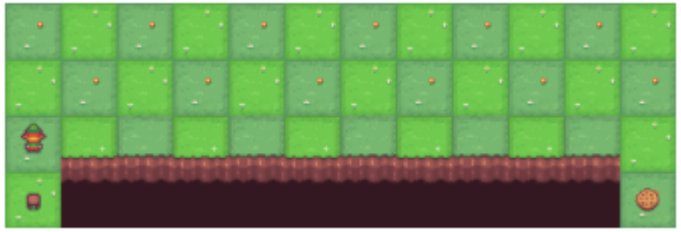
```
-17.0
```

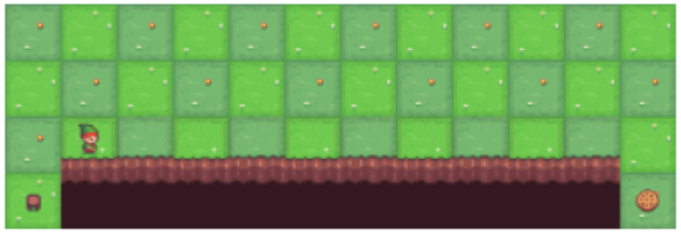
## 1-7. Increase Noises (10 pts)

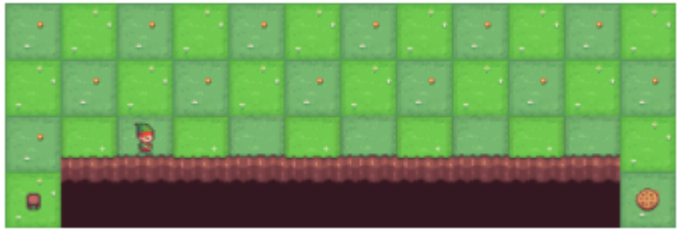
Increase noise and see how your obtained policy changes. Do this for both above algorithms and repeat above steps.

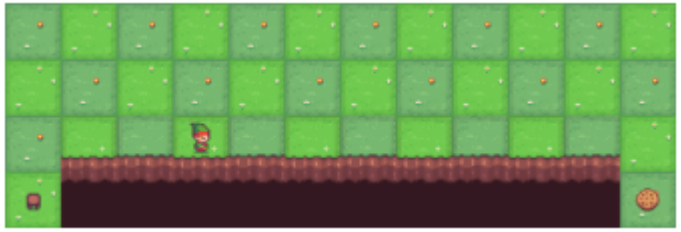
In [28]:
```
# ######################### YOUR CODE HERE ##########################
cliff_walker_ql = CliffWalkerQL(env, 0.35)
alpha = 0.8
gamma = 0.9
epsilon = 0.2
episodes = 1000
cliff_walker_ql.learn(episodes, alpha, gamma, epsilon)
print(cliff_walker_ql.evaluate(episodes))
print("*****************************************************")
cliff_walker_sarsa = CliffWalkerSARSA(env, 0.5)
alpha = 0.1
gamma = 0.9
epsilon = 0.2
episodes = 3000
cliff_walker_sarsa.learn(episodes, alpha, gamma, epsilon)
print(cliff_walker_sarsa.evaluate(episodes))
# ######################### END YOUR CODE ###########################
# مخص است مقدار  ریوارد هر دو منفی تر می شود  زیرا نویز داریم و این اختلاف برای کیو لرنینگ به طبع بیشتر نیز است
# و هر چه مقدار نویز بیشتر باشد به طبع، دیرتر به مقصد می رسیم.
```

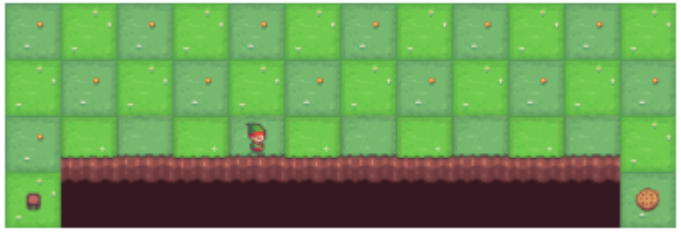


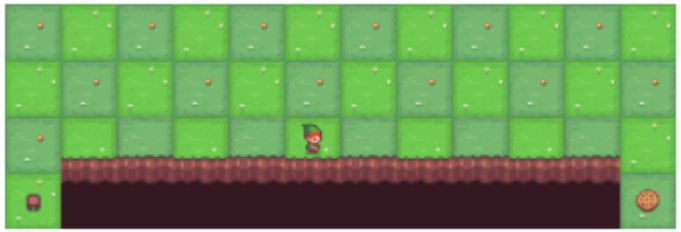


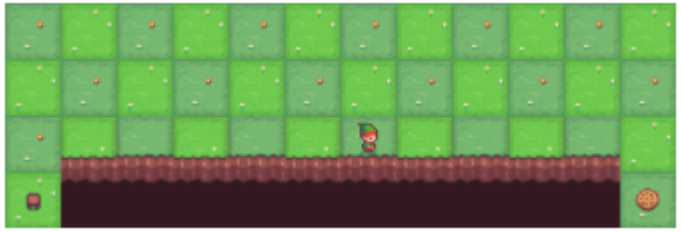


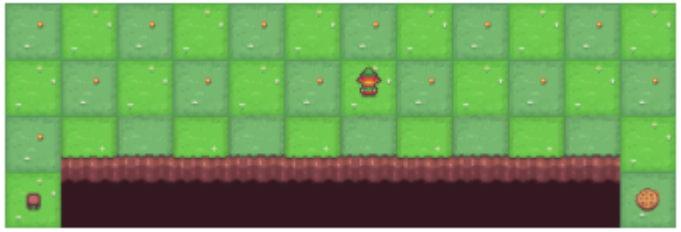


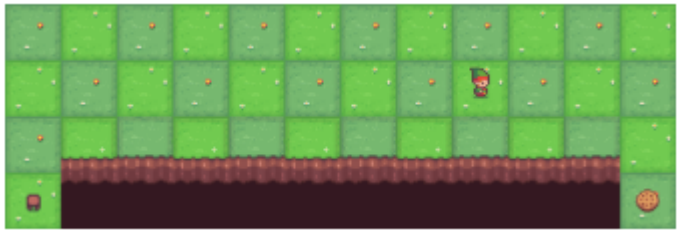


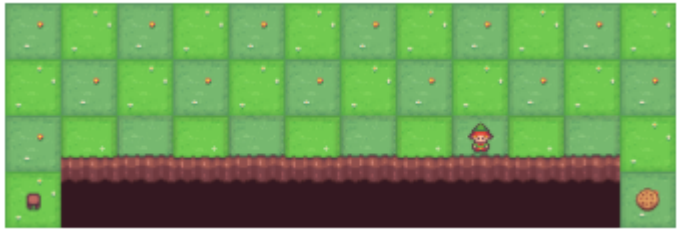


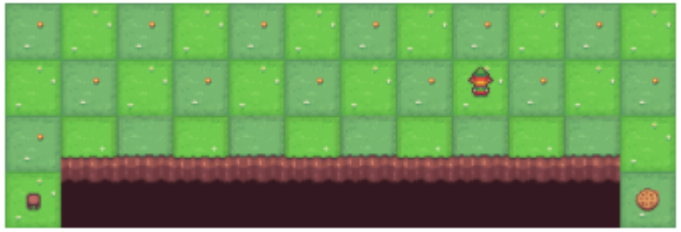


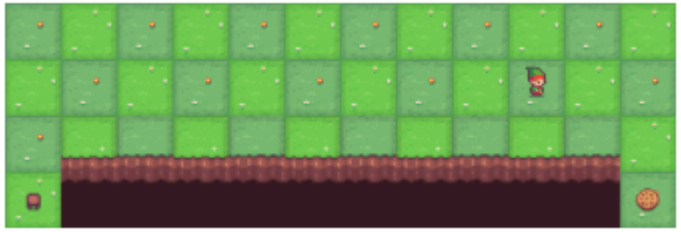


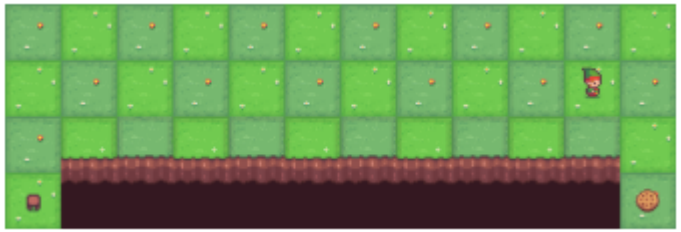


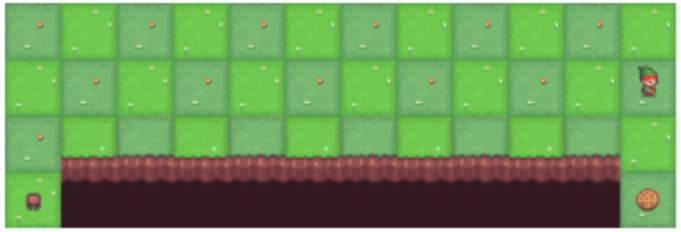


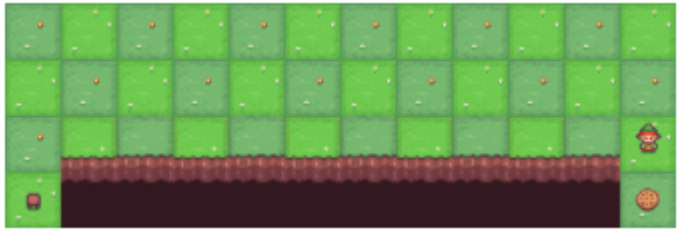


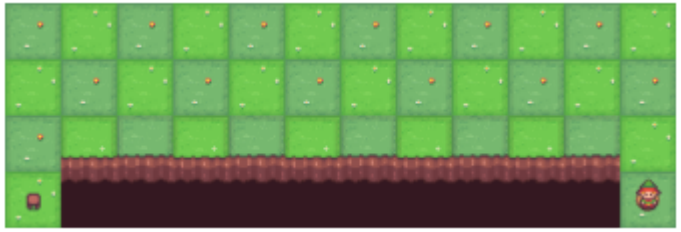




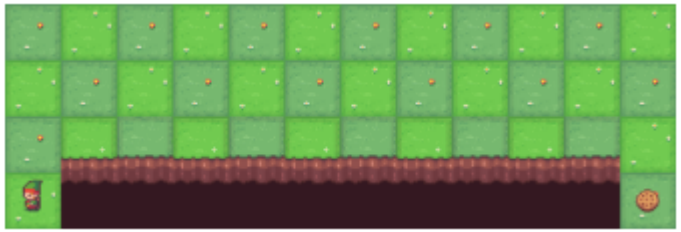


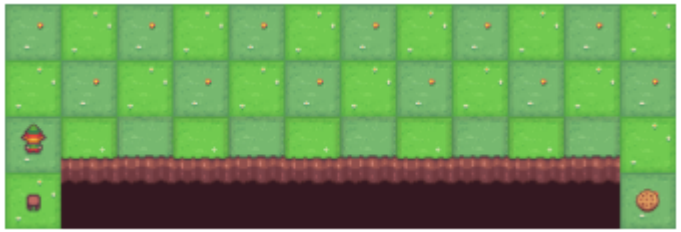


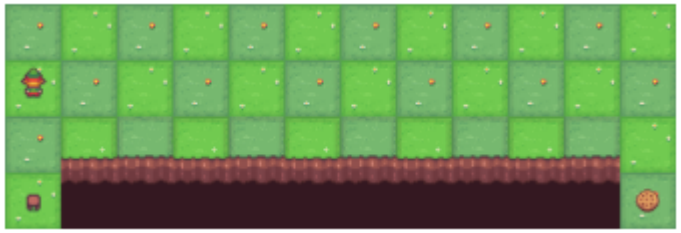


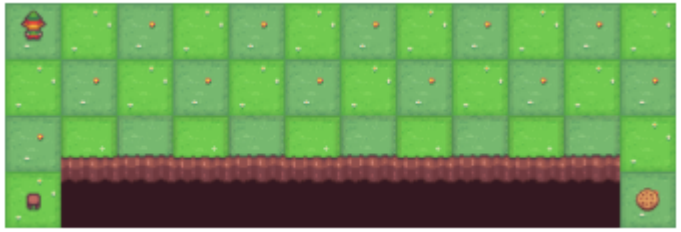


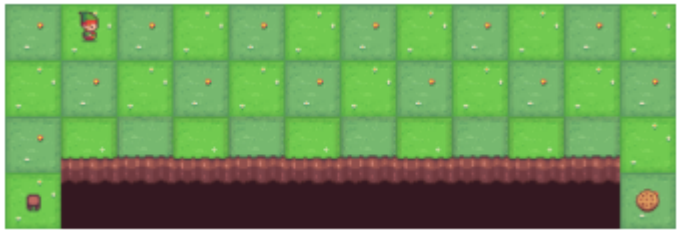
```
-154.466
**********************************************************
```

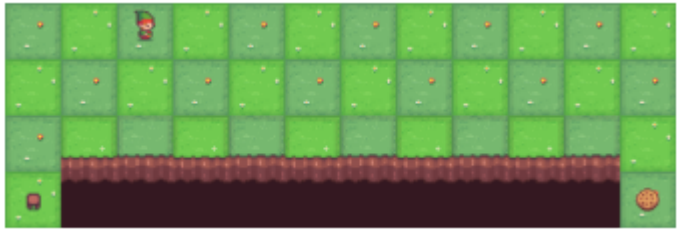


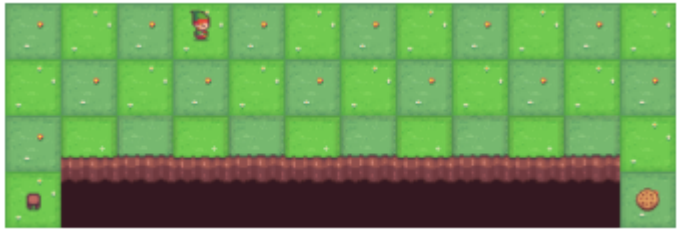


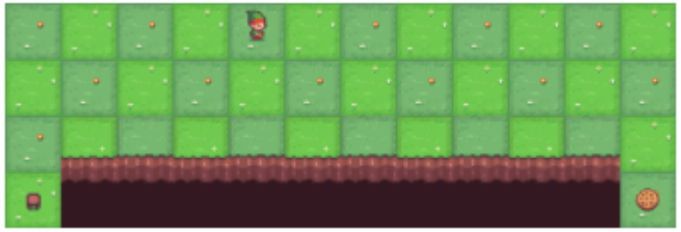


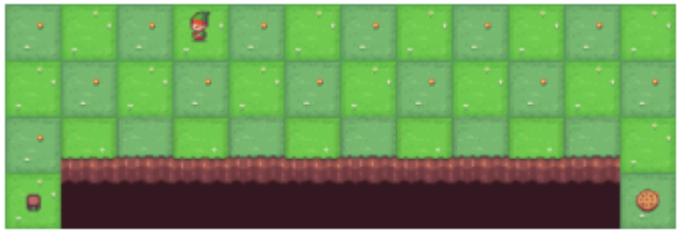


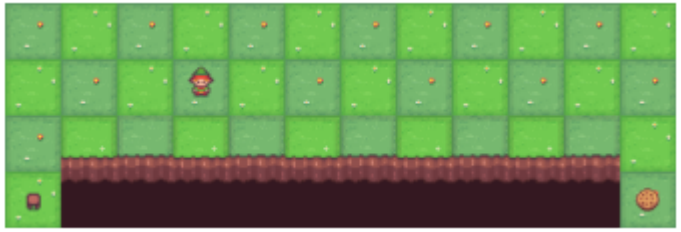


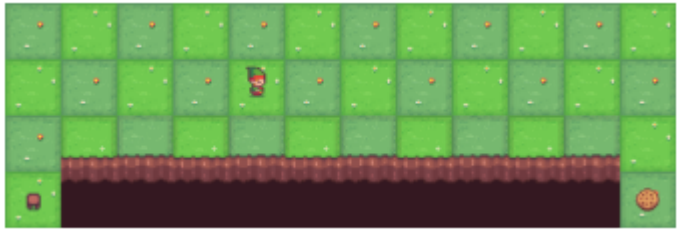


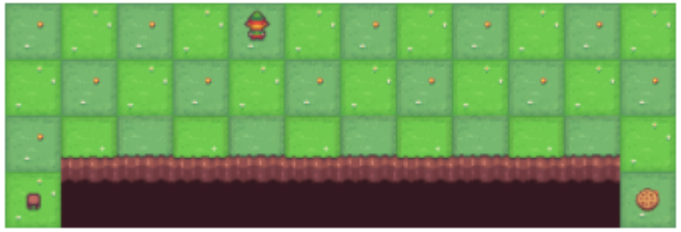


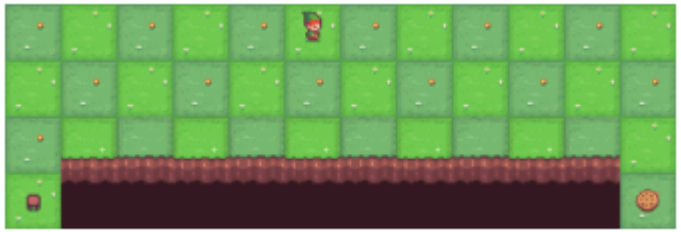


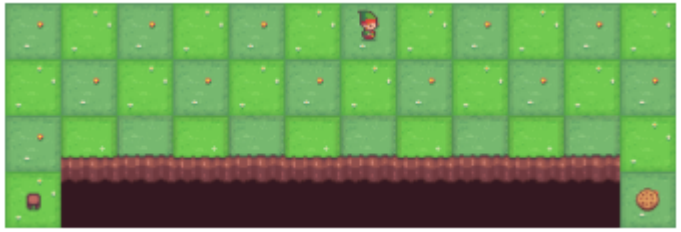


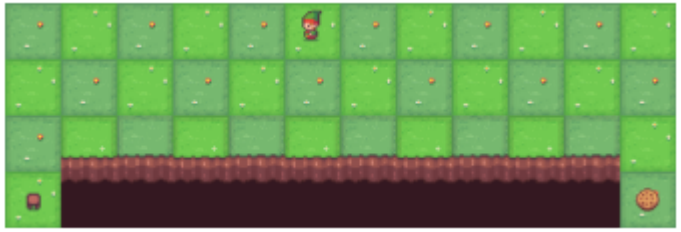


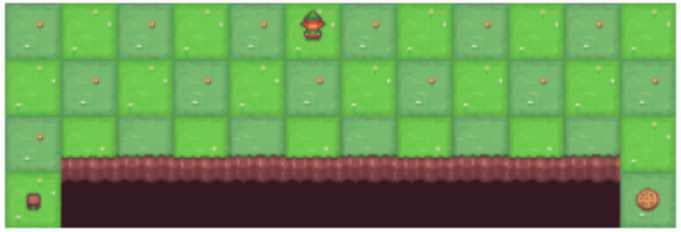


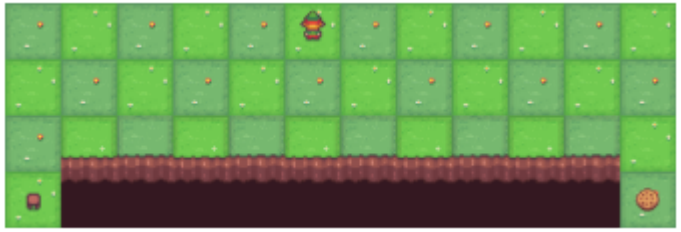


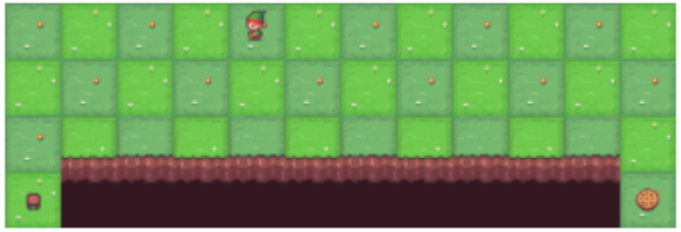


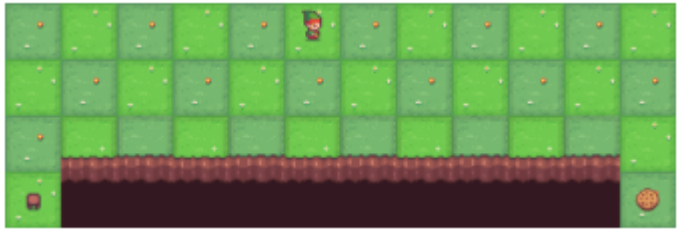


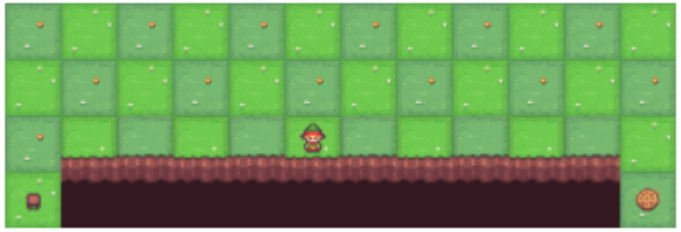


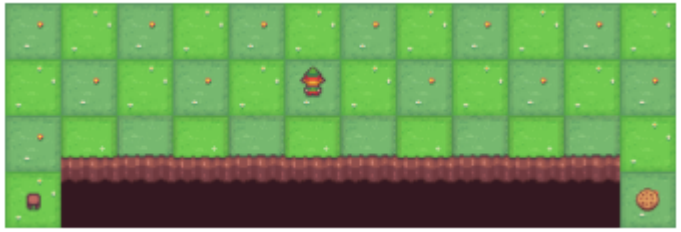


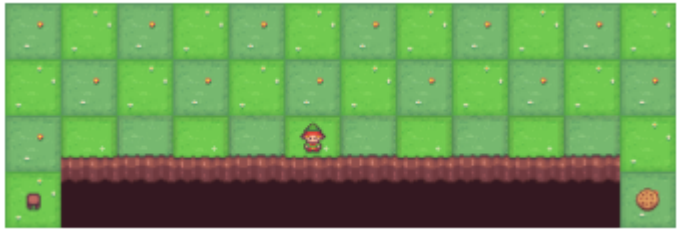


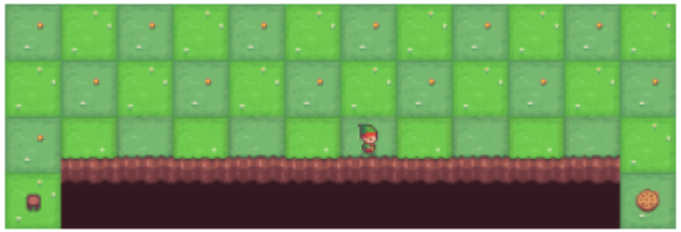


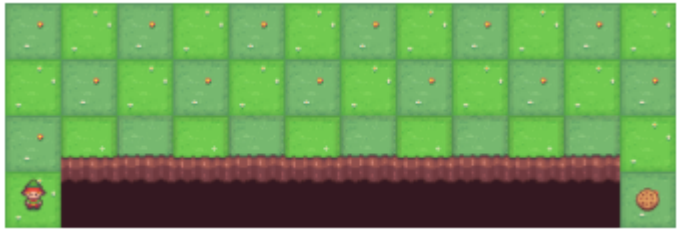


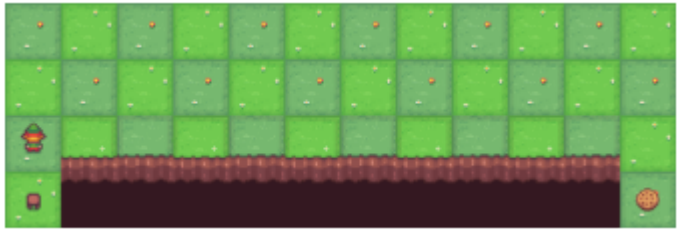


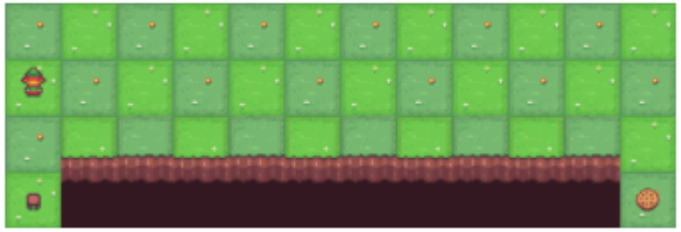


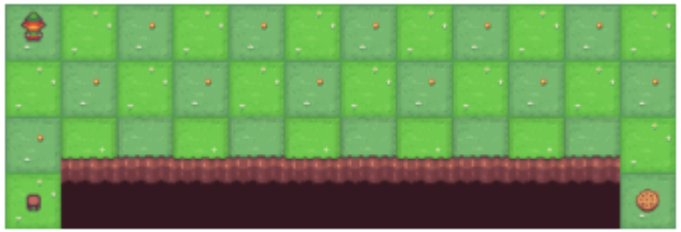


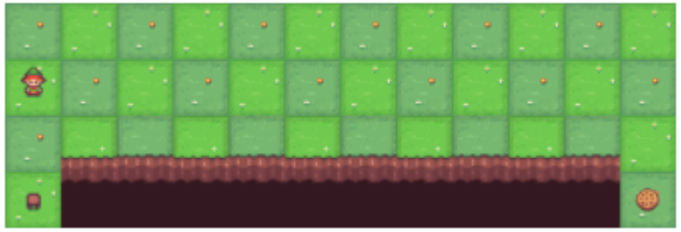


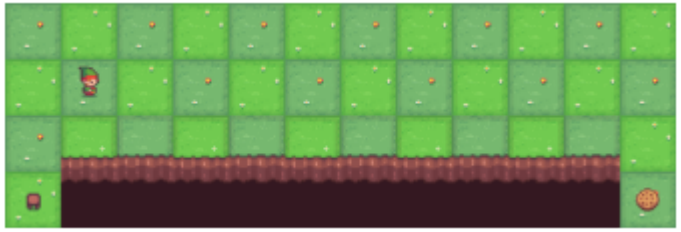


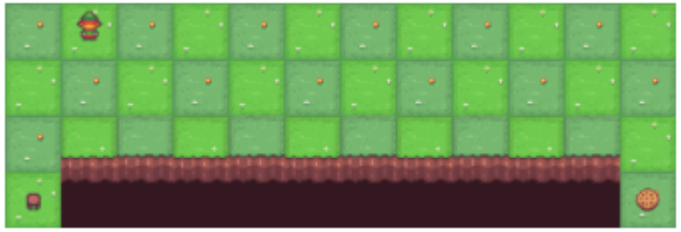


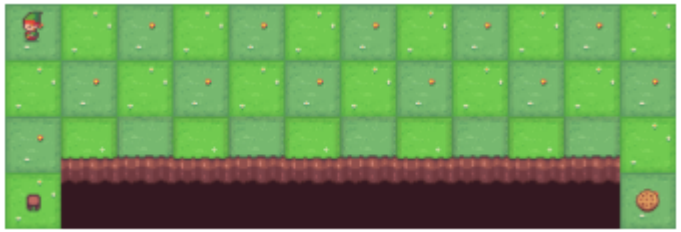


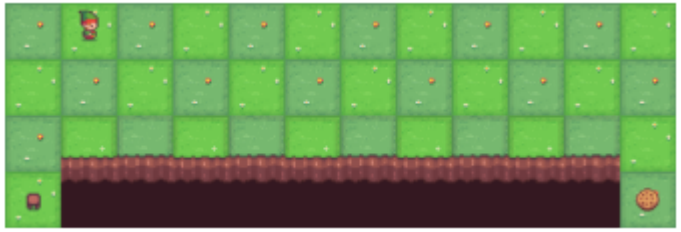


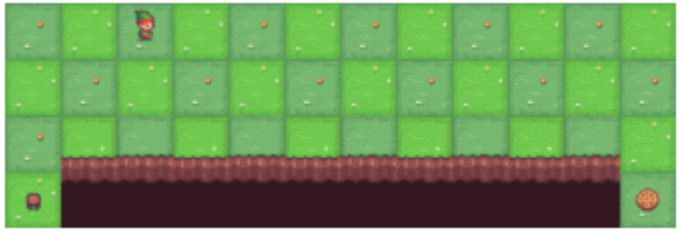


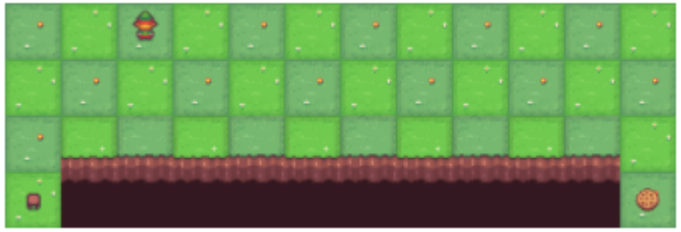


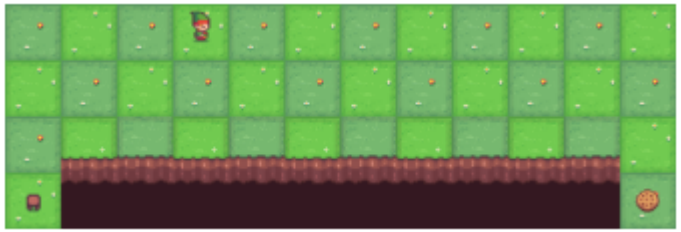


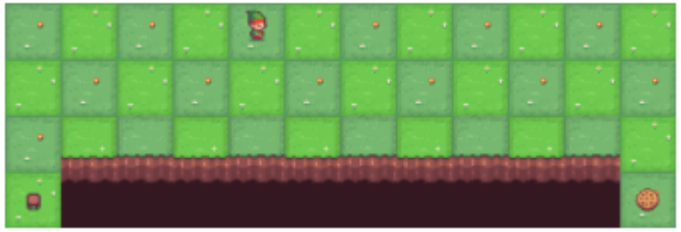


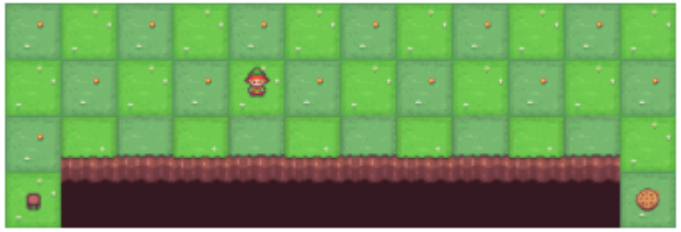


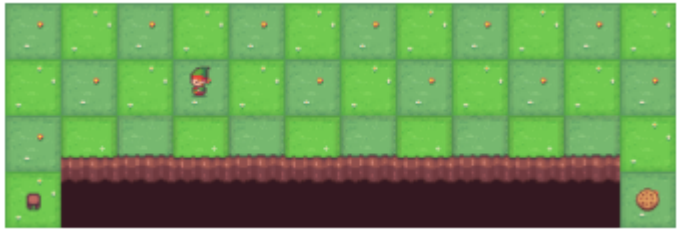


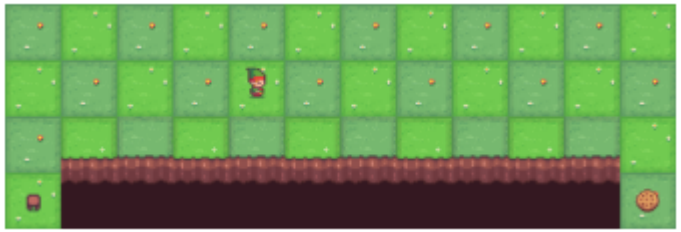


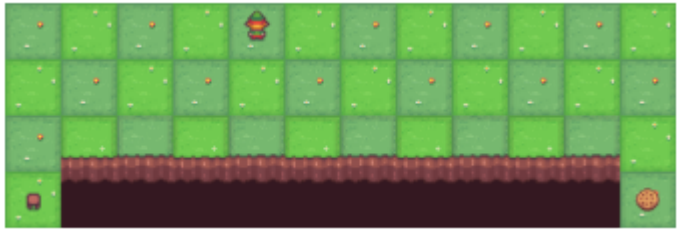


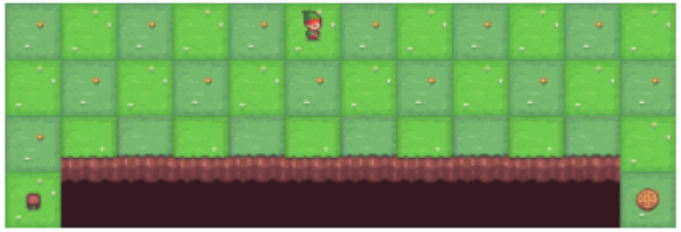


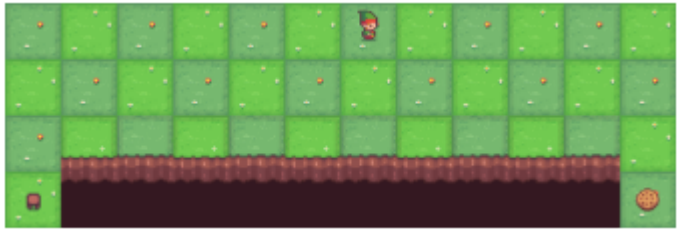


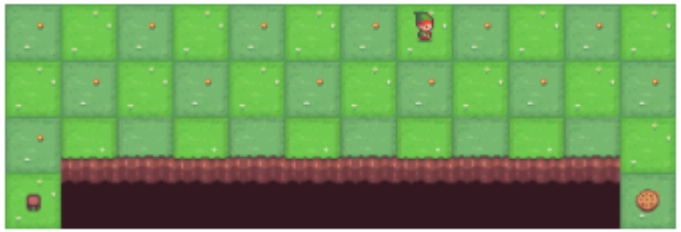


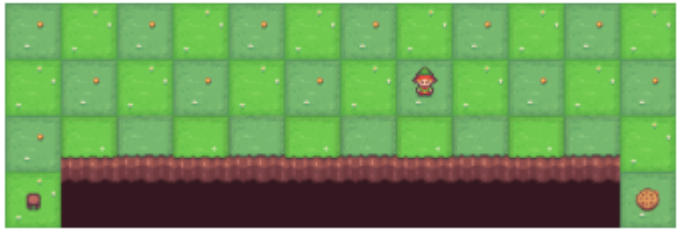


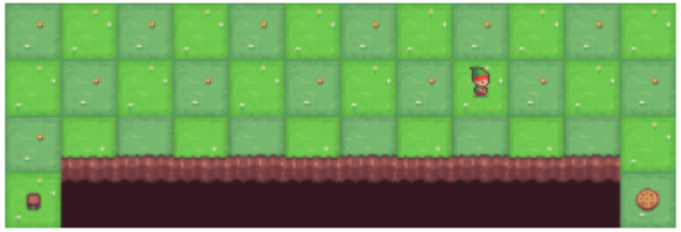


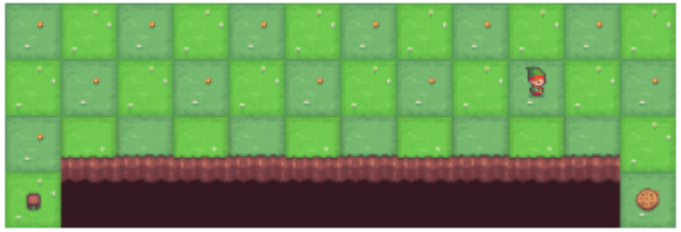


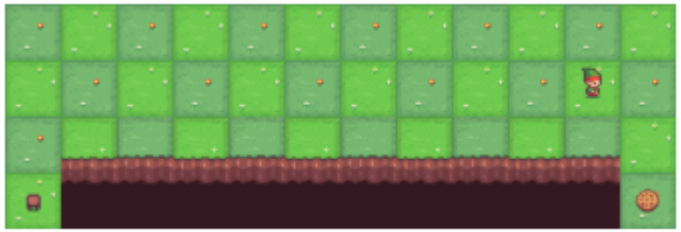


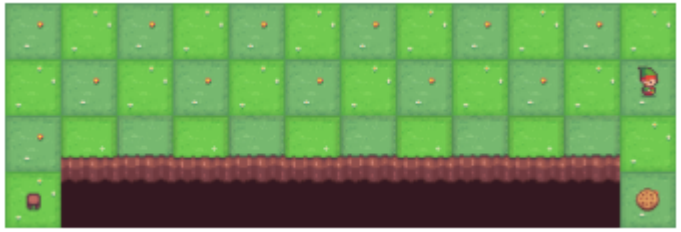


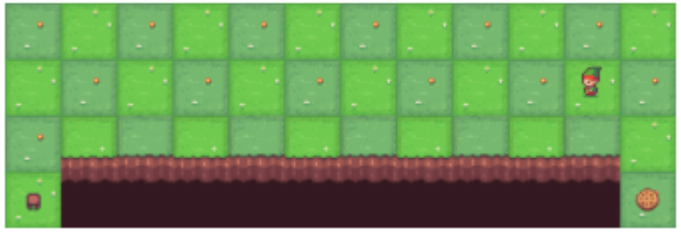


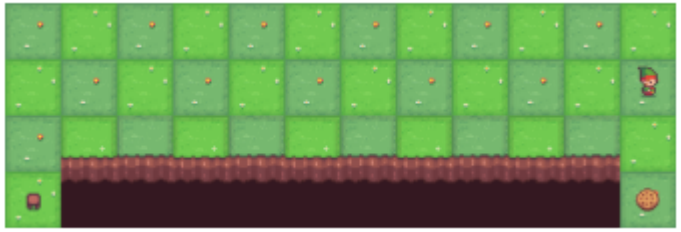


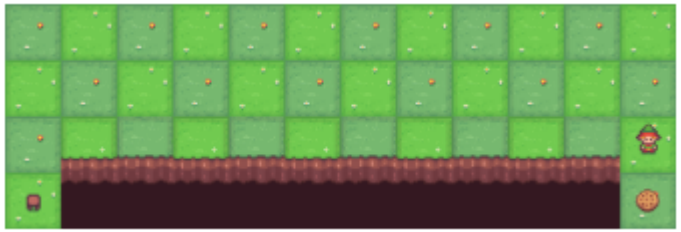


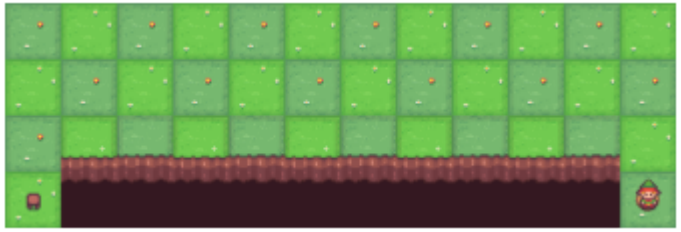












```
-89.59566666666667
```

# 2. Taxi Driver (30 pts)

The next game that we want to implement is **Taxi Driver**.

# 2-1. Environment

In [15]:
```
env = gym.make('Taxi-v3', render_mode='rgb_array')
spec = gym.spec('Taxi-v3')

print(f"Action Space: {env.action_space}")
print(f"Observation Space: {env.observation_space}")
print(f"Max Episode Steps: {spec.max_episode_steps}")
print(f"Nondeterministic: {spec.nondeterministic}")
print(f"Reward Range: {env.reward_range}")
print(f"Reward Threshold: {spec.reward_threshold}\n")
```

```
Action Space: Discrete(6)
Observation Space: Discrete(500)
Max Episode Steps: 200
Nondeterministic: False
Reward Range: (-inf, inf)
Reward Threshold: 8
```

In [16]:
```
Actions =  {0: 'DOWN',
            1: 'UP',
            2: 'RIGHT',
            3: 'LEFT',
            4: 'Pickup passenger',
            5: 'Drop off passenger'}
```

You can read more about the game and its observation space in this link (https://gymnasium.farama.org/environments/toy_text/taxi/#observation-space).

## 2-2. Q-Learning (5 pts)

Implement Q-Learning algorithm for this problem. (Of course, you can use the code you implemented in the previous section and just enjoy the result (: )

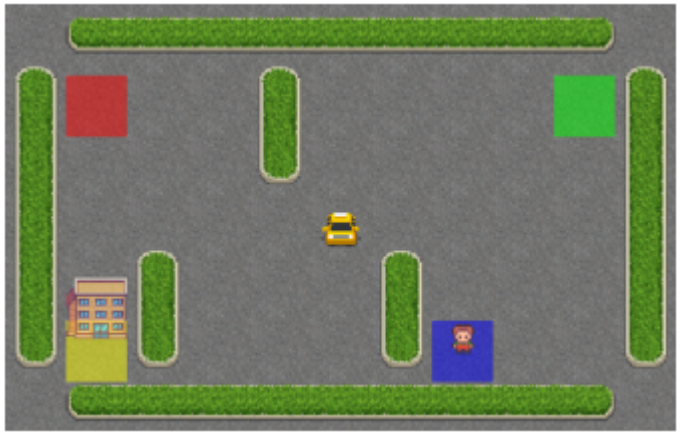
In [17]:
```
class TaxiQL(Agent):
    def learn(self, num_episodes, alpha, gamma, epsilon):
        ######################### YOUR CODE HERE ##########################
        self.q_values = np.zeros((500, 6))

        for i in range(num_episodes):
            state = self.env.reset()[0]
            terminated = False
            while not terminated:
                action = self.create_policy(state, epsilon, 6)
                new_state, reward, terminated = self.act(action)
                self.q_values[state][action] = self.q_values[state][action] +\
                                               alpha * (reward + gamma * np.max(self.q_values[r
                                                        self.q_values[state][action])
                state = new_state
        ######################### END YOUR CODE ###########################
```

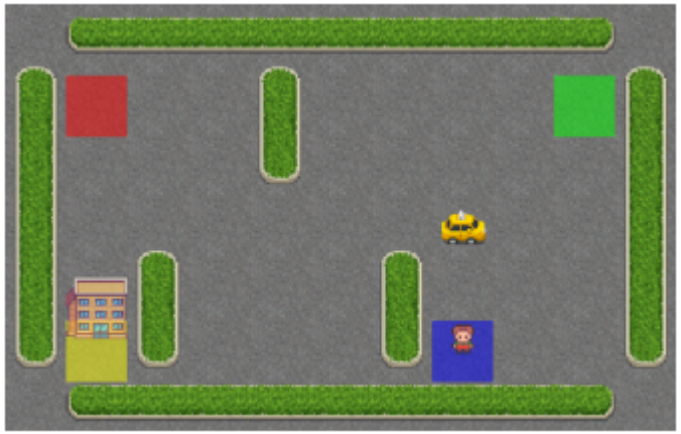
## 2-3. Q-Learning Evaluation (5 pts)

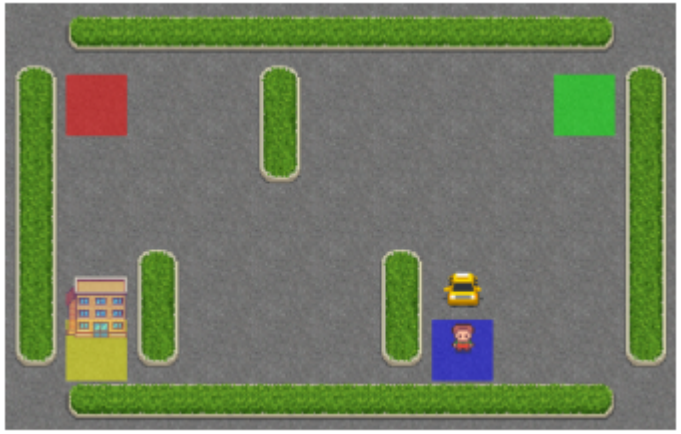
Train your agent two times, once with 1000 episodes and once with 10000 episodes. Then evaluate it and display the result. Using the `visualize` function, show the path your agent takes in one of the episodes.

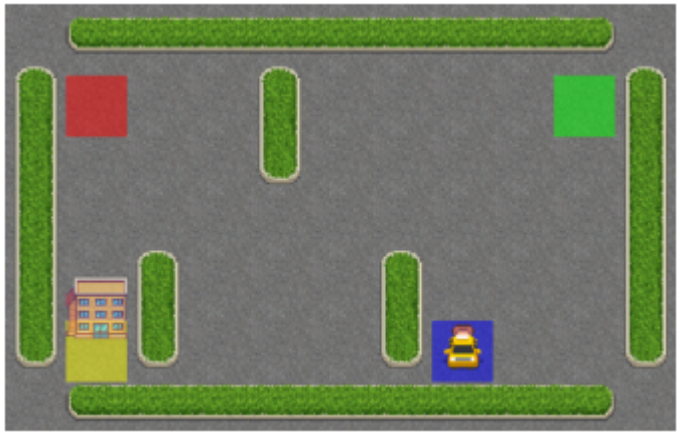
In [18]:
```
taxi_ql = TaxiQL(env, 0)
alpha = 0.9
gamma = 0.95
epsilon = 0.1
episodes = 3000
taxi_ql.learn(episodes, alpha, gamma, epsilon)
```

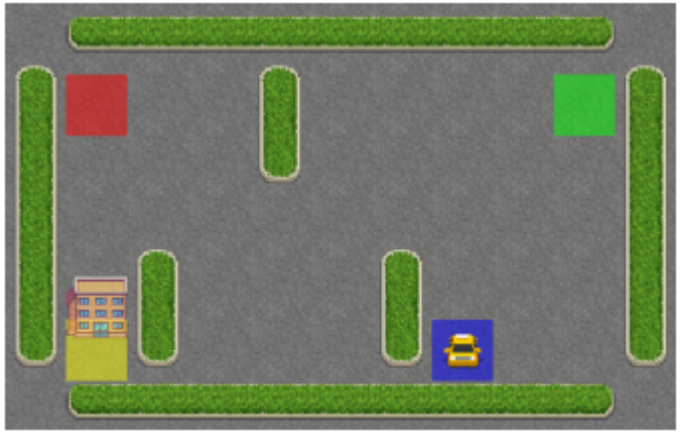
In [19]:
```
######################### YOUR CODE HERE ##########################
taxi_ql.evaluate(episodes)
########################## END YOUR CODE ##########################
```

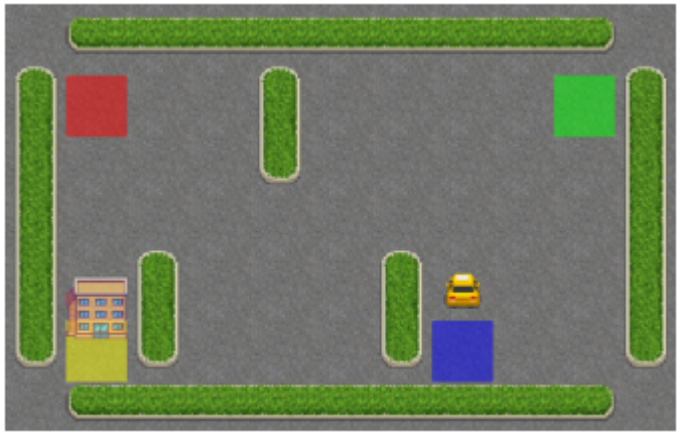


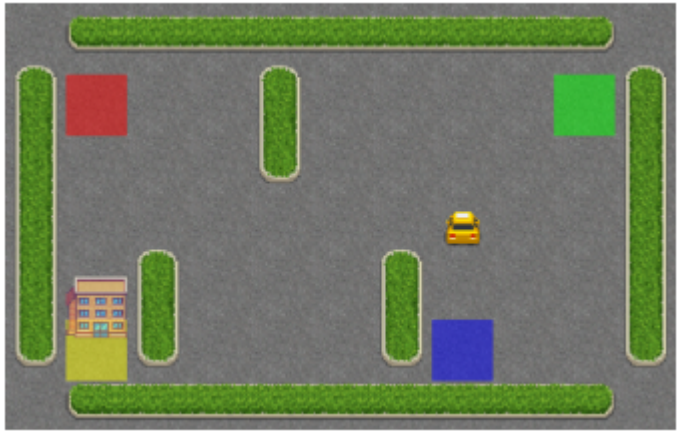


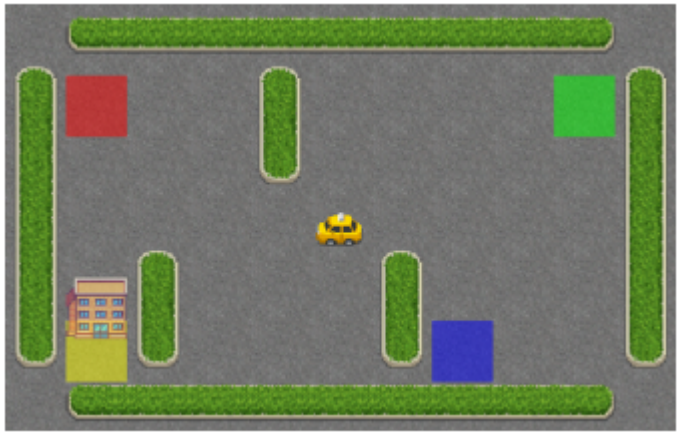


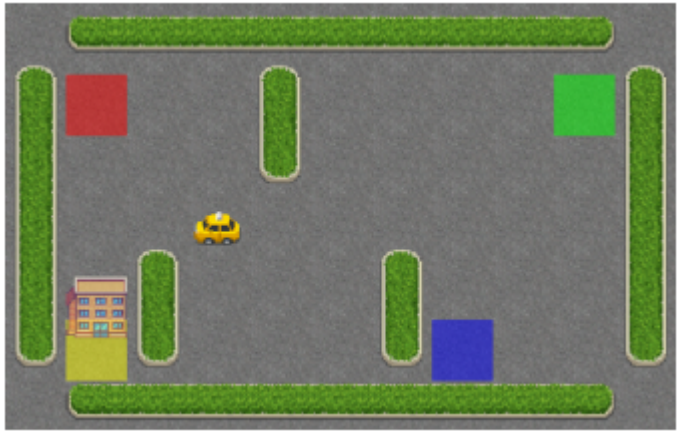


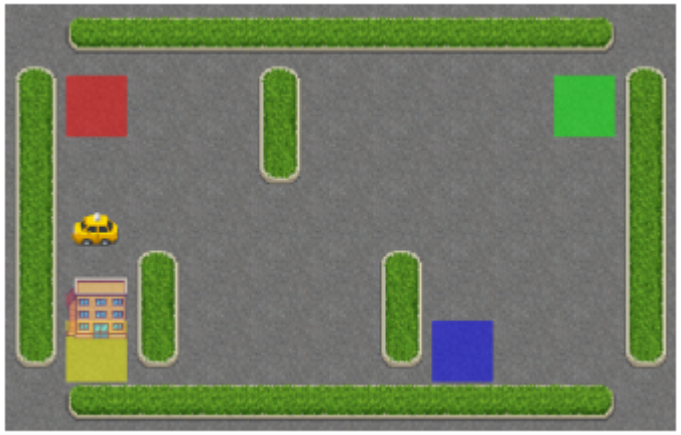


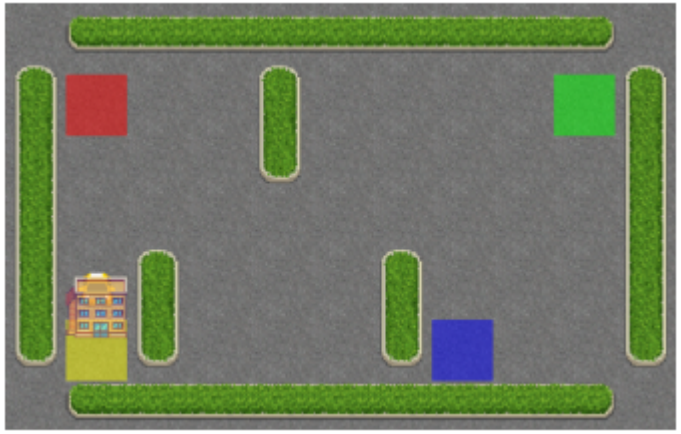


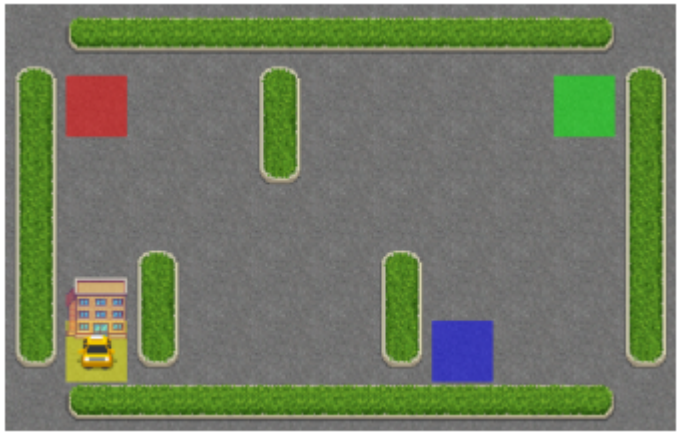


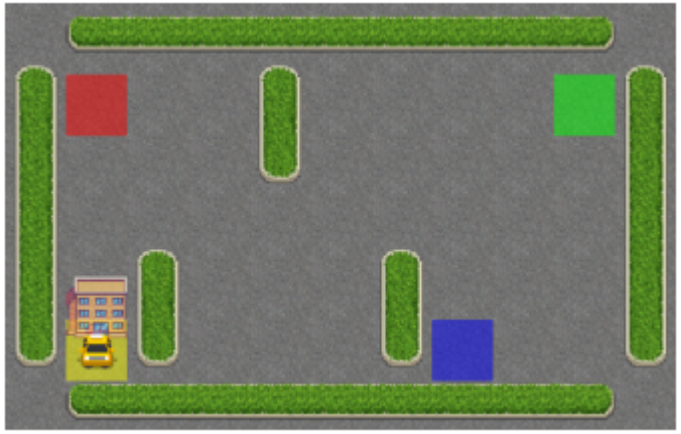








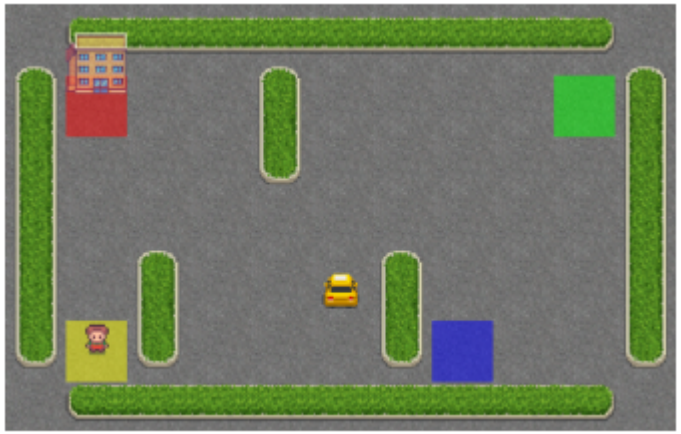


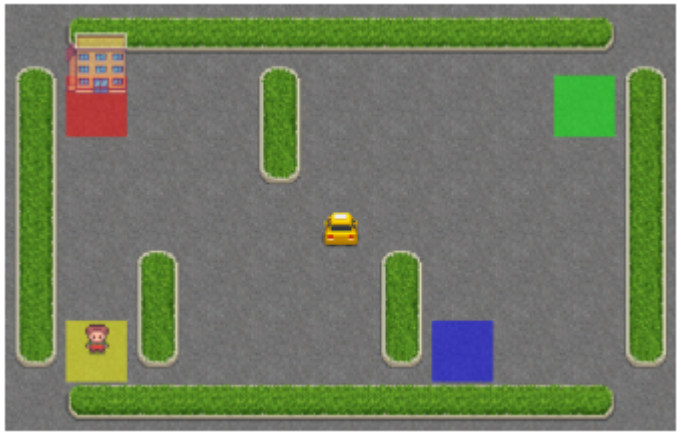


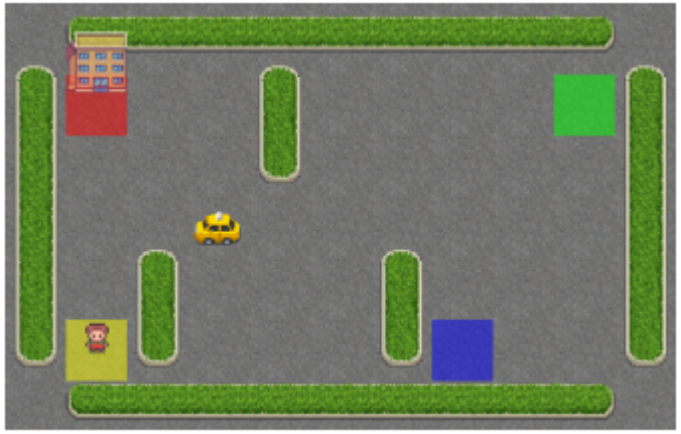
```
7.815333333333333
```

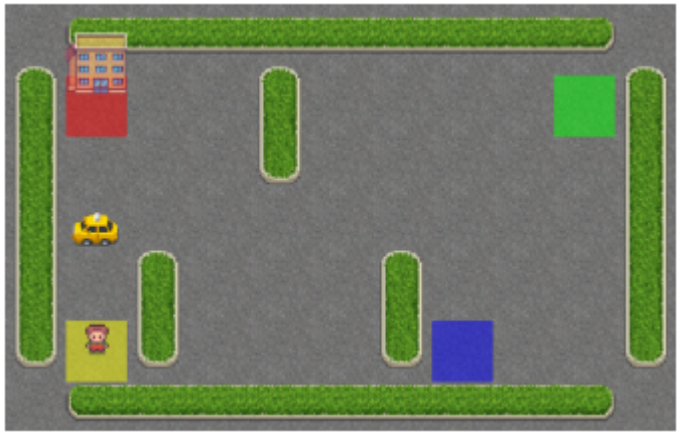
In [20]:
```
taxi_ql = TaxiQL(env, 0)
alpha = 0.8
gamma = 0.95
epsilon = 0.1
episodes = 10000
taxi_ql.learn(episodes, alpha, gamma, epsilon)
```

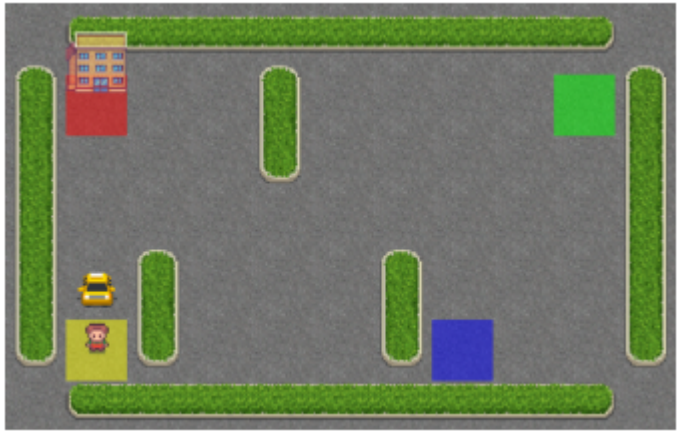
In [21]:
```
########################## YOUR CODE HERE ##########################
taxi_ql.evaluate(episodes)
########################## END YOUR CODE ###########################
```

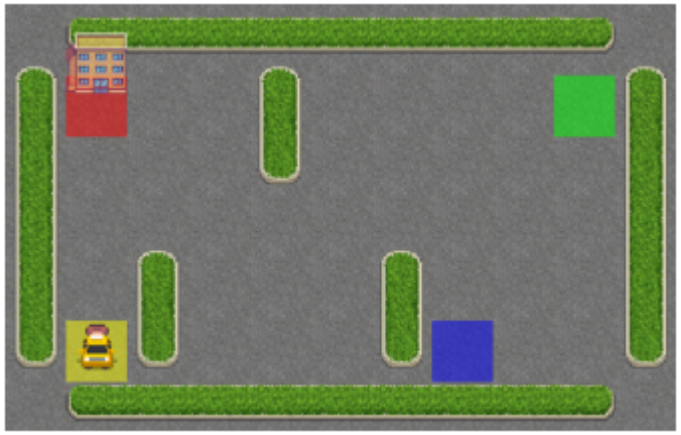


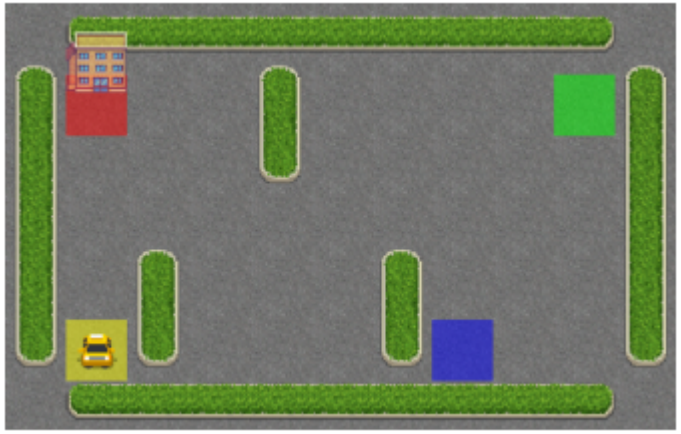


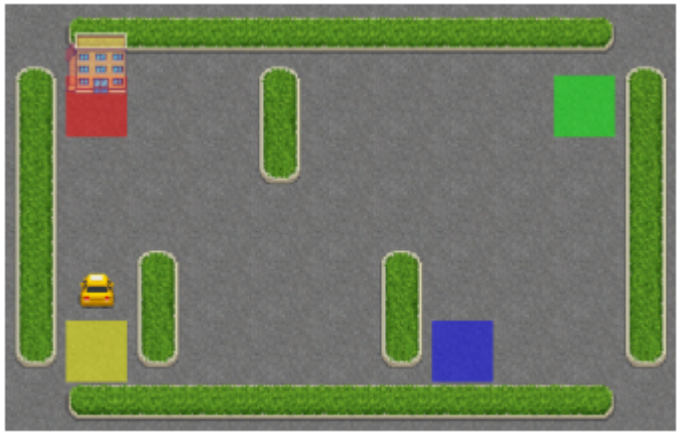


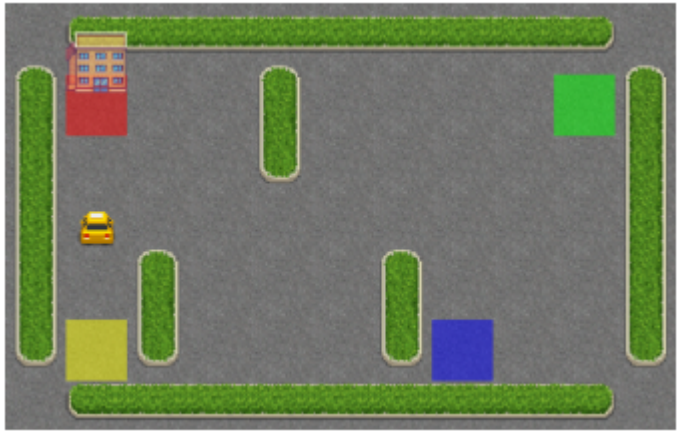


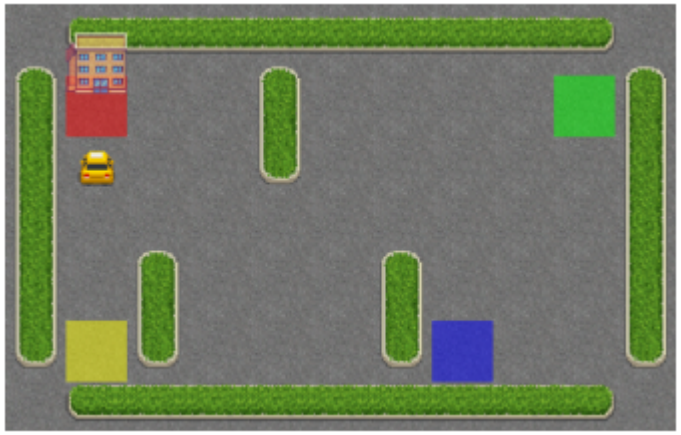


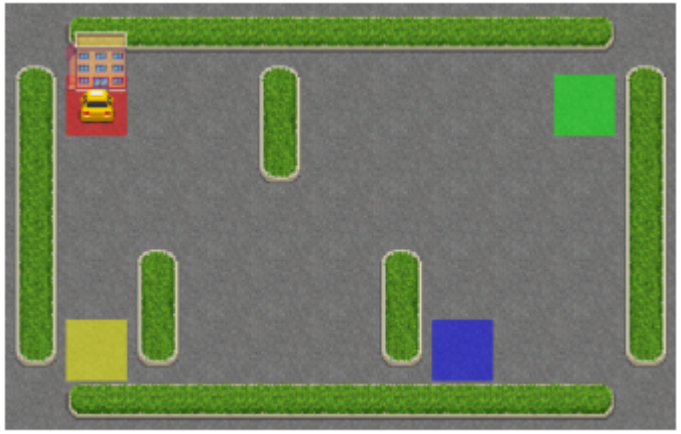


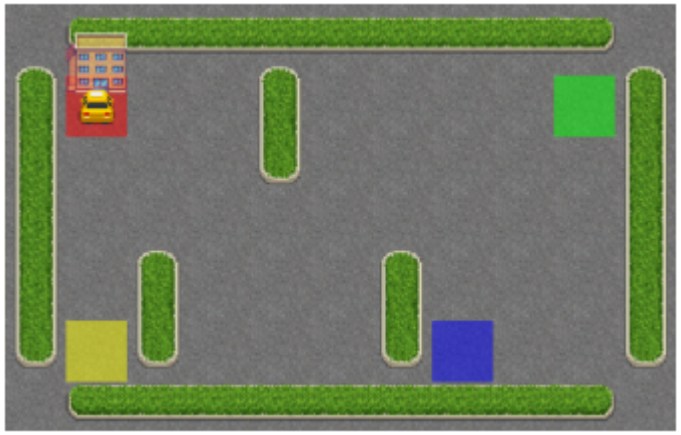












```
7.9187
```

## 2-4. TD(2) (15 pts)

In this section, you have to implement the Taxi problem using TD(2) algorithm. The difference between this method and the previous methods is in the number of movements that we look from the future.

For example, the SARSA TD(2) update rule can be represented as:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left( R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) + \gamma^2 Q(s_{t+2}, A_{t+2}) - Q(S_t, A_t) \right)$$

Where $Q(S_t, A_t)$ is the current estimate of the expected return for taking action $A_t$ in state $S_t$, $\alpha$ is the learning rate, $R_{t+1}$ is the reward received after taking action $A_t$ in state $S_t$, $\gamma$ is the discount factor, and $Q(S_{t+1}, A_{t+1})$ is the estimated return for taking action $A_{t+1}$ in the next state $S_{t+1}$.

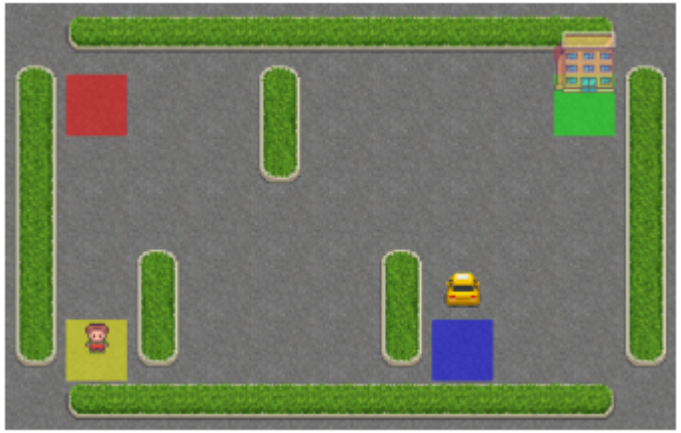
You can see this video (https://youtu.be/AJiG3ykOxmY) for more details about TD methods.

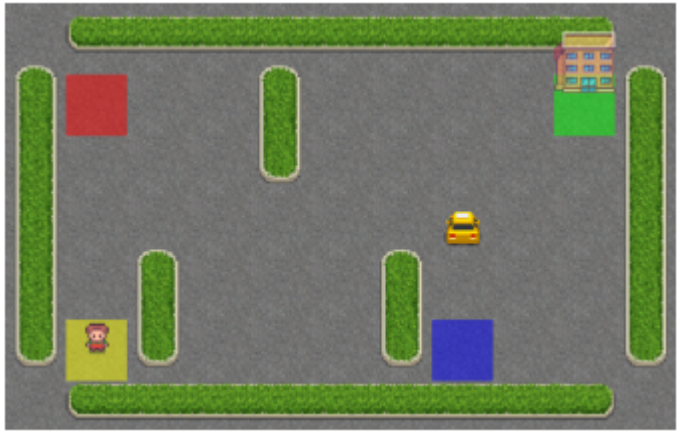
In [22]:
```
class TaxiTD2(Agent):

    def learn(self, num_episodes, alpha, gamma, epsilon):
        ######################### YOUR CODE HERE ########################
        self.q_values = np.zeros((500, 6))
        for i in range(num_episodes):
            state = self.env.reset()[0]
            terminated = False
            action = self.create_policy(state, epsilon, 6)
            second_state, reward1, terminated = self.act(action)
            second_action = self.create_policy(second_state, epsilon, 6)
            while not terminated:
                third_state, reward2, terminated = self.act(second_action)
                third_action = self.create_policy(third_state, epsilon, 6)
                self.q_values[state][action] = self.q_values[state][action] +\
                                               alpha * (reward1 + (gamma * (self.q_values[secor
                                                       ((gamma * gamma) * (self.q_values[thir(
                                                       self.q_values[state][action])

                state = second_state
                action = second_action
                second_state = third_state
                second_action = third_action
                reward1 = reward2
        ######################### END YOUR CODE #########################
```

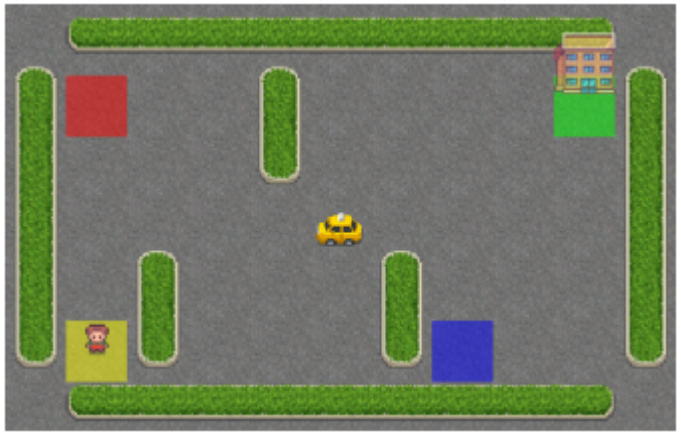
# 2-5. TD(2) Evaluation (5 pts)

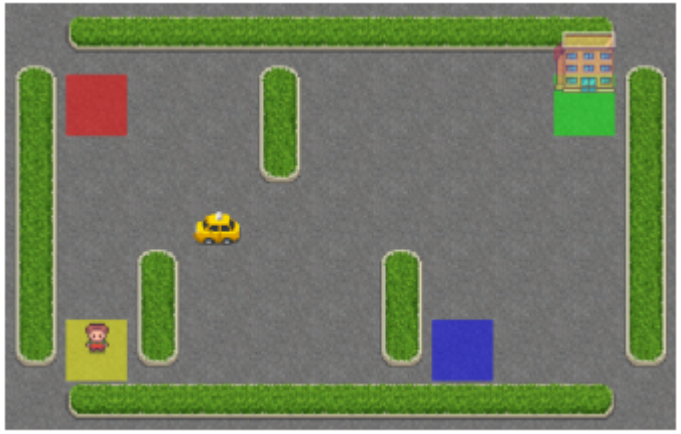
In [23]:
```
taxi_td2 = TaxiTD2(env, 0)
alpha = 0.864
gamma = 0.27
epsilon = 0
episodes = 1000
taxi_td2.learn(episodes, alpha, gamma, epsilon)
```

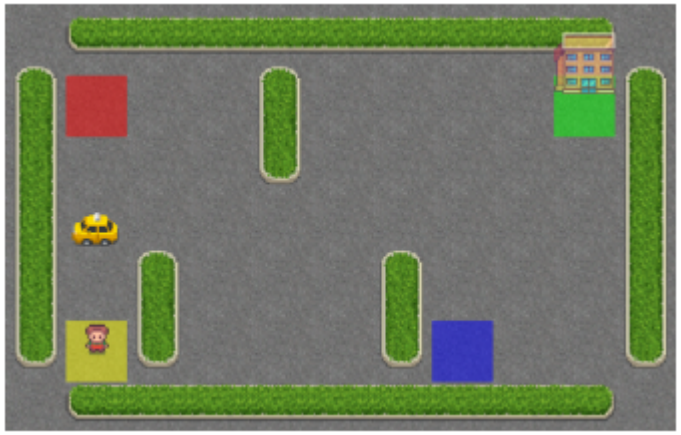
In [24]:
```
######################### YOUR CODE HERE ##########################
print(taxi_td2.evaluate(episodes))
taxi_td2 = TaxiTD2(env, 0)
alpha = 0.864
gamma = 0.27
epsilon = 0
episodes = 9000      # more episodes to learn better and get the better reward
taxi_td2.learn(episodes, alpha, gamma, epsilon)
print(taxi_td2.evaluate(episodes))
########################## END YOUR CODE ###########################
```

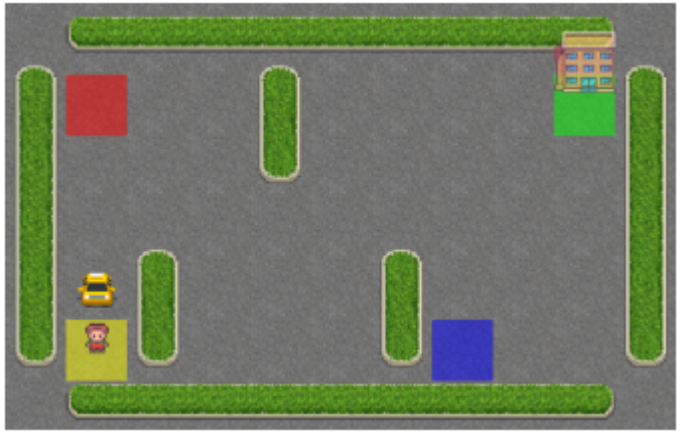


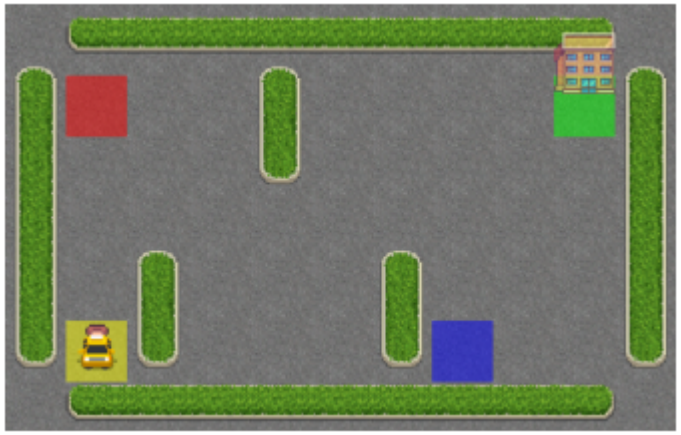


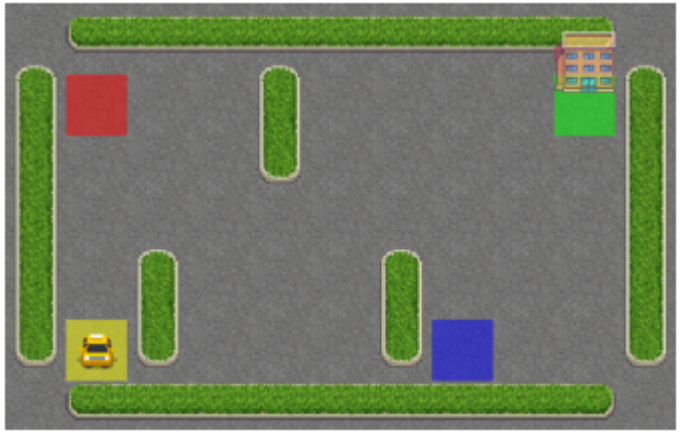


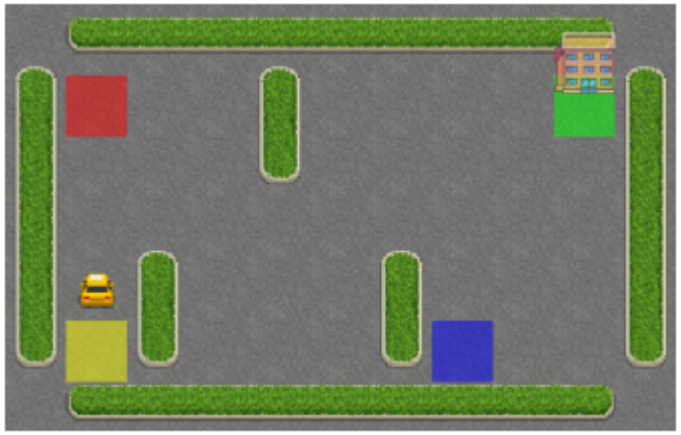


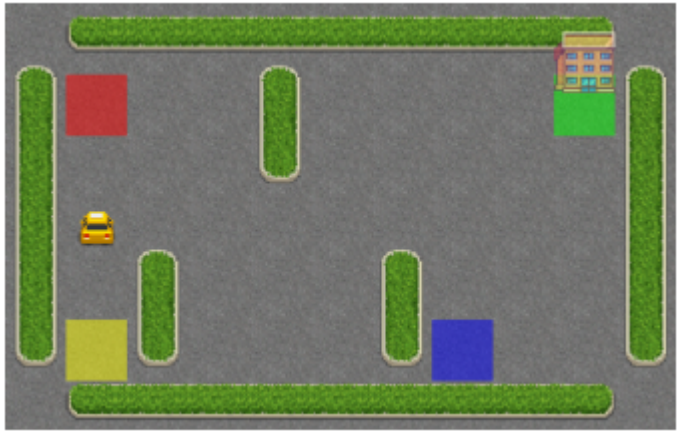


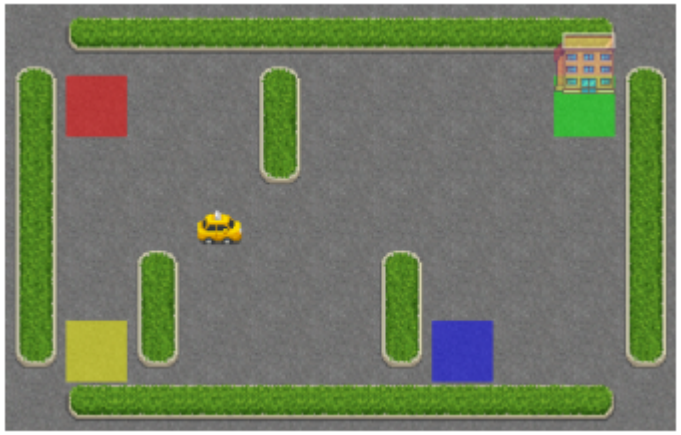


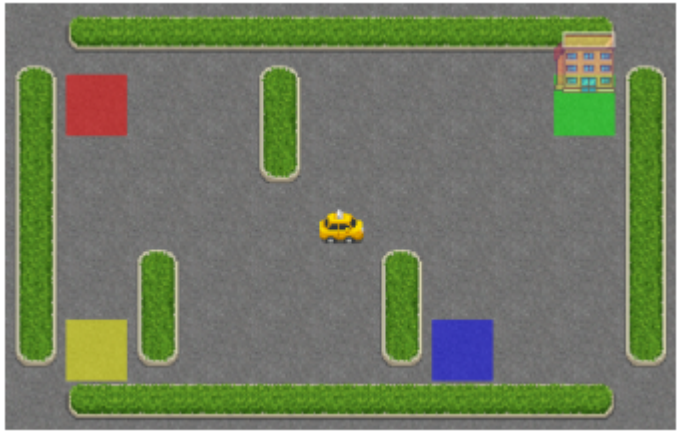


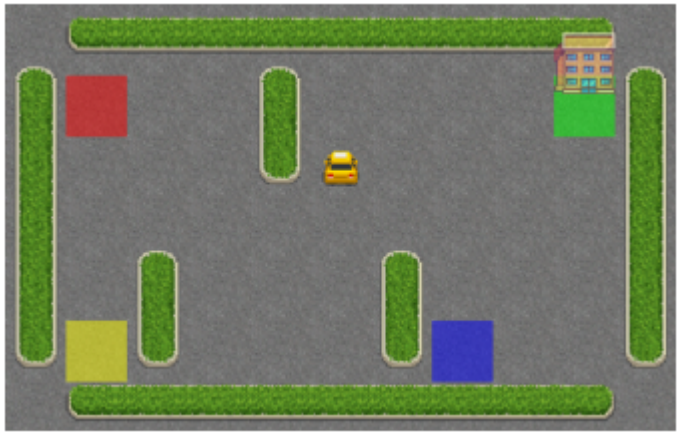


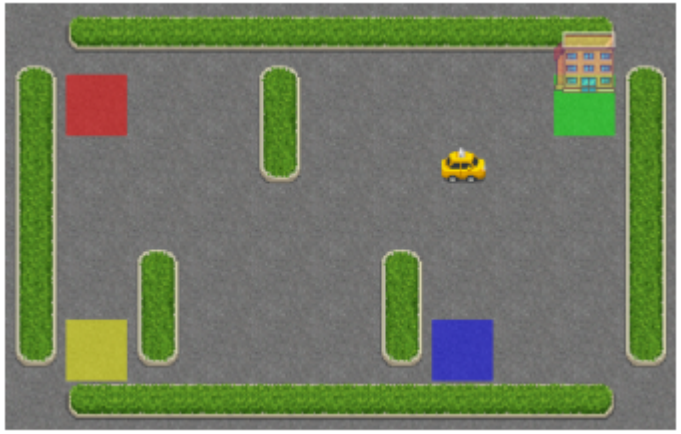


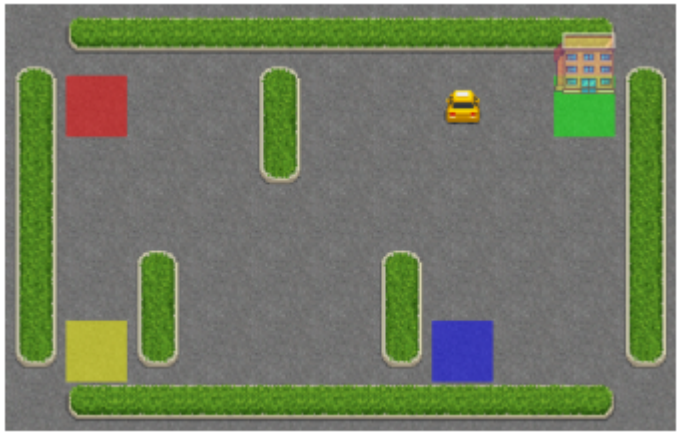


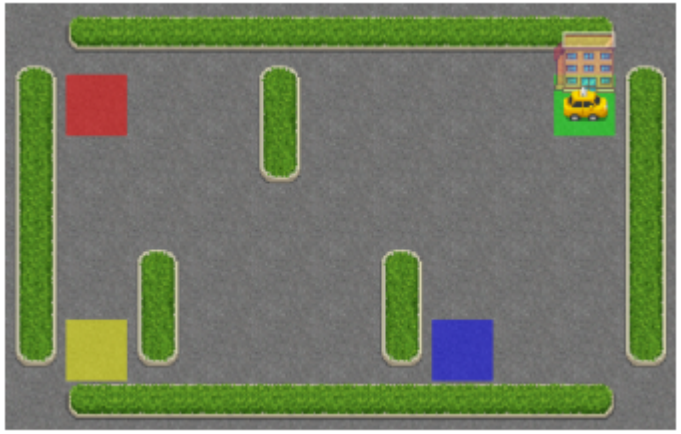


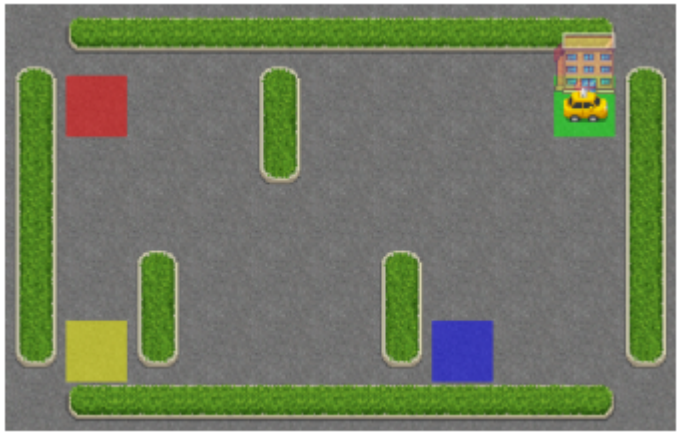




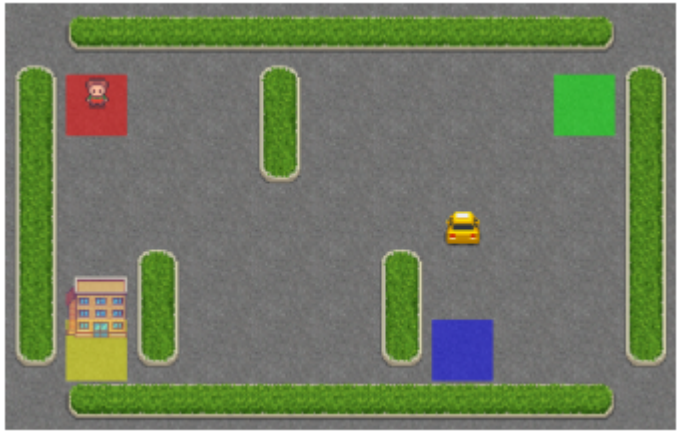


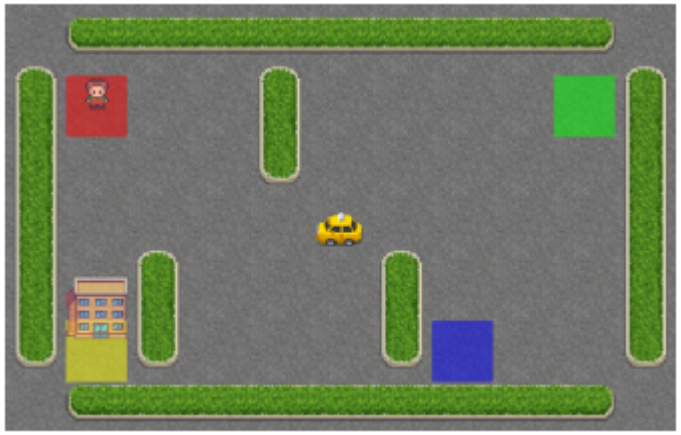


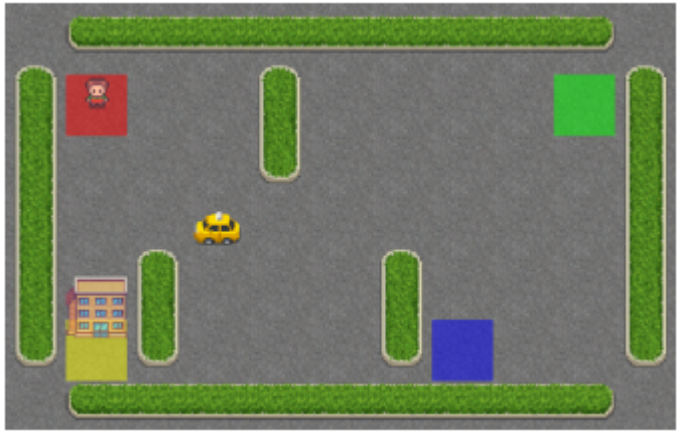


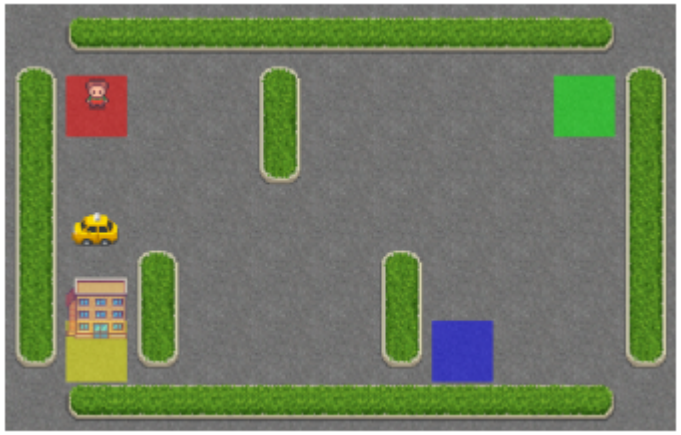


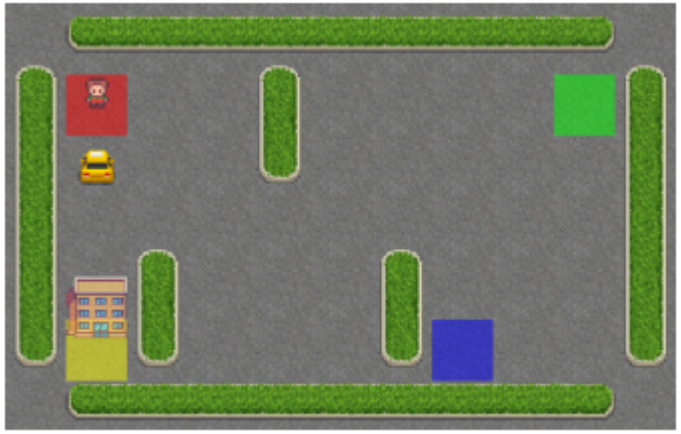
```
2.5
```

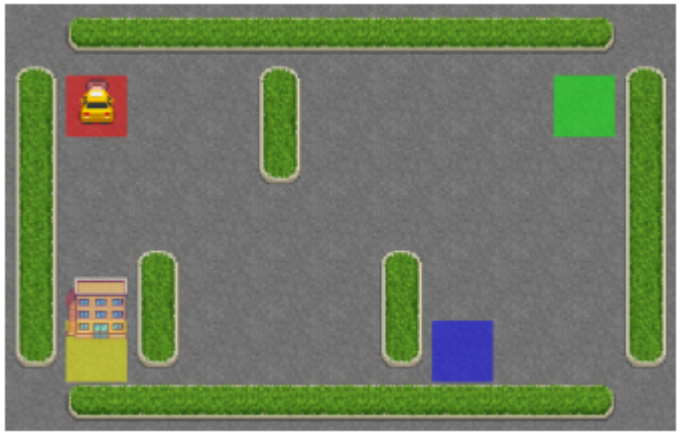


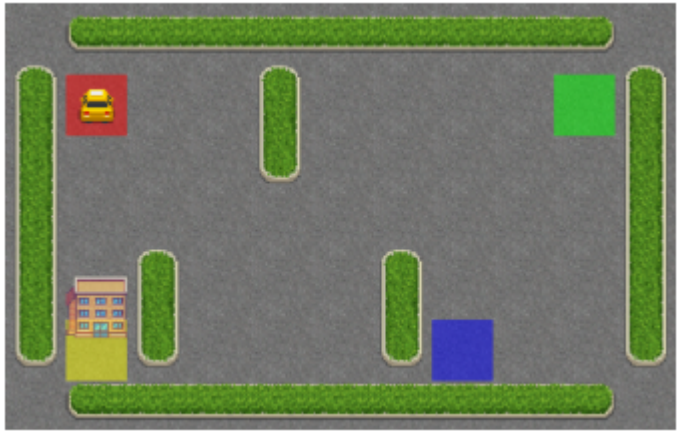


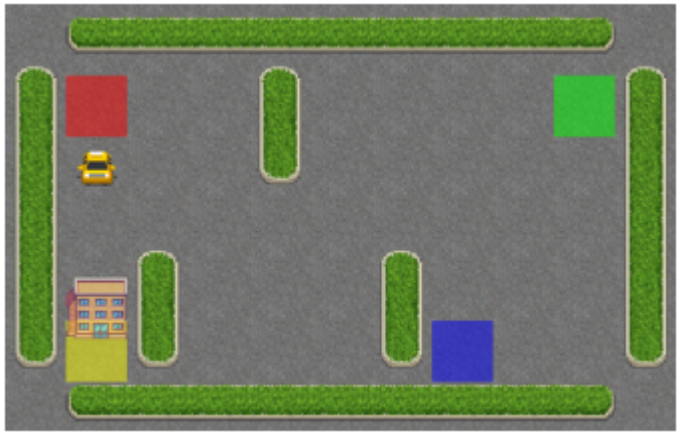


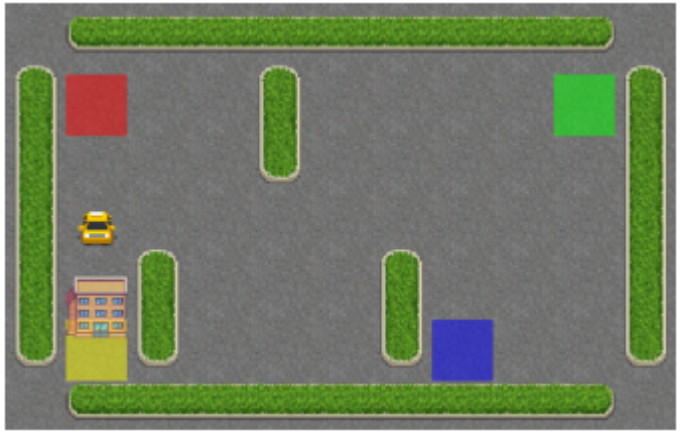


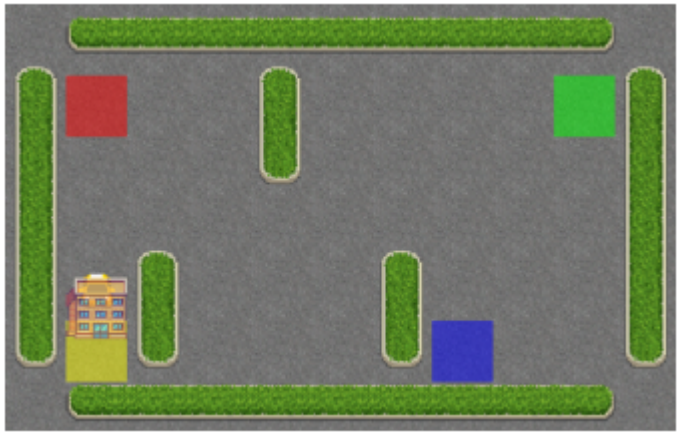


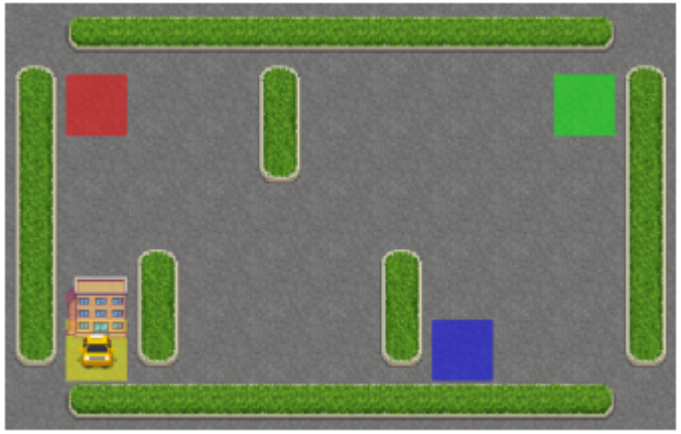


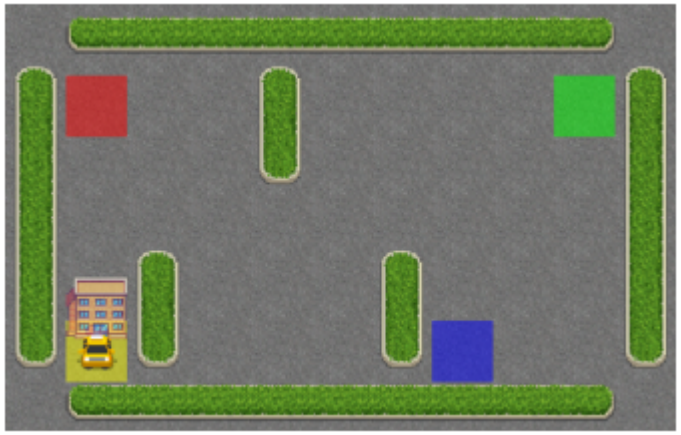












```
7.9591111111111115
```

| | |
|---|---|
| | You can see that the TD(2) method can reach an acceptable policy faster. Try to explain why. |
| | QL learns the optimal policy while SARSA learns a near optimal policy. QL is a more aggressive agent, while SARSA is more conservative.In walking near the cliff, QL will take the shortest path because it is optimal, while SARSA will take the longer but safer route (to avoid unexpected falling).<br>But TD(2) that the number of movements which we look from the future is more, can reach a policy |

faster because it's SARSA TD(2) and it checks the Q_values of
next states in each update and it follows the current policy. So it's faster.
But if we give epsilon more than zero, the random movements increases and we get the negative
reward too