Problem Set 3: Reinforcement Learning and Constraint Satisfaction

Q1 Reinforcement Learning [Total: 40 points]

Q1a. [15 points] *Written RL problem*

For a simple cliff-walker Q-value problem, compute the Q-values at each state. The goal is the cell marked in green (with a reward of 0), and stepping on the red cells results in immediate failure with reward -100. All other states get a reward of -1.

The Q-value equation is given by:

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

Assume a discount factor of 1.0 (i.e. $\gamma = 1$). As an example, Q-values for one cell have been computed for you.

(R = -1) U: D: L: R:	(R = -1) U: D: L: R:	(R = -1) U: D: L: R:	(R = -1) U: D: L: R:
(R = -1) U: D: L: R:	(R = -1) U: D: L: R:	(R = -1) U: D: L: R:	(R = -1) U: -2.0 D: 0.0 L: -2.0 R: -1.0
(R = -1) U: D: L: R:	Cliff (R = -100)		Goal (R = 0)

(R = -1) U: - 5 D: - 4 L: - 5 R: - 4	(R = -1) U: - 4 D: - 3 L: - 5 R: - 3	(R = -1) U: - 3 D: - 2 L: - 4 R: - 2	(R = -1) U: - 2 D: - 1 L: - 3 R: - 2
(R = -1) U: - 5 D: - 5 L: - 4 R: - 3	(R = -1) U: - 4 D: - 100 L: - 4 R: - 2	(R = -1) U: - 3 D: - 100 L: - 3 R: - 1	(R = -1) U: -2.0 D: 0.0 L: -2.0 R: -1.0
(R = -1) U: - 4 D: - 5 L: - 5 R: - 100	Cliff (R = -100)		Goal (R = 0)

Q1b. [25 points] *Coding RL problem*

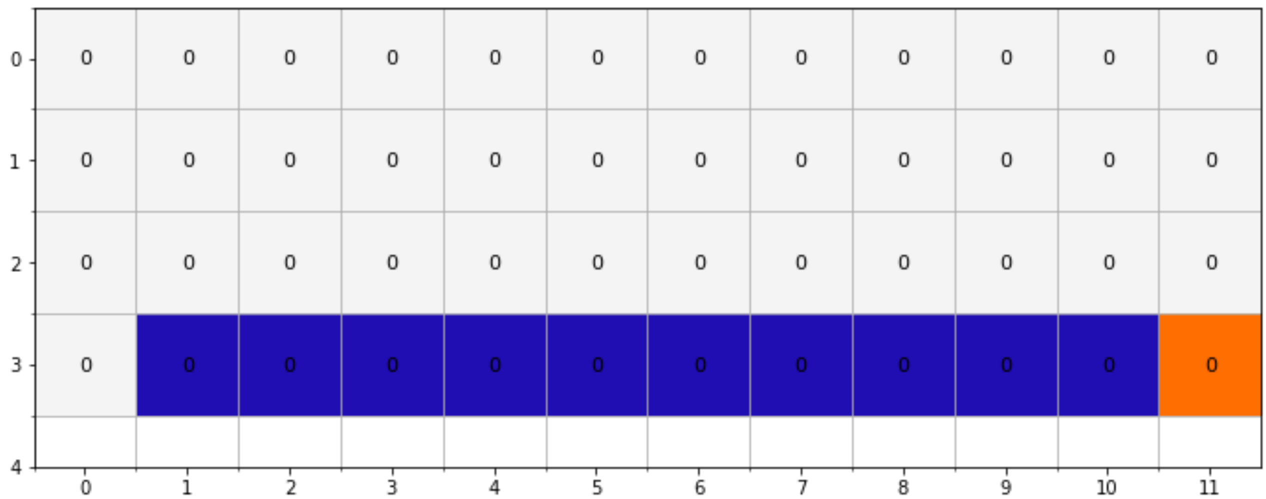
Let's start by reading about the Cliff Walking Problem

In [5]:
```
import numpy as np
import matplotlib.pyplot as plt
from CliffWalker import GridWorld
```

We create a 4×12 grid, similar to the written problem in 1a. above on which you will implement a Q-learning algorithm.

In [6]:
```
env = GridWorld()

# The number of states in simply the number of "squares" in
num_states = 4 * 12
# We have 4 possible actions, up, down, right and left
num_actions = 4
```



Tasks

We ask you to implement two functions:

- an ϵ-greedy action picker
- a basic Q-learning algorithm

ϵ-greedy choices make the greedy choice most of the time but choose a random action ϵ fraction of the time. For example, for $\epsilon = 0.1$, if a random number is ≤ 0.1, then a random action is taken.

In [28]:
```
def egreedy_policy(q_values, state, epsilon=0.1):
    '''
    Choose an action based on a epsilon greedy policy.
    A random action is selected with epsilon probability, el
    '''
    decider = np.random.random()
    if (decider <= epsilon):
        return np.random.choice(4)
    else:
        temp = None
        index = None
        qs = q_values[state]
        for i in range(len(qs)):
            if temp == None:
                index = i
                temp = qs[i]
            elif qs[i] >= temp:
                index = i
                temp = qs[i]
        return index
```

1

Now, you can implement a basic Q-learning algorithm. For your reference, use the following:

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
    $S \leftarrow S'$
  until S is terminal

We provide a skeleton code, leaving the Q-value update for you to implement.

Note: learning rate α, exploration rate ϵ, and discount factor γ are provided as inputs to the function

In [29]:

```
def q_learning(env, num_episodes=200, render=True, epsilon=0
               learning_rate=0.5, gamma=0.9):
    q_values = np.zeros((num_states, num_actions))
    ep_rewards = []

    for _ in range(num_episodes):
        state = env.reset()
        done = False
        reward_sum = 0

        while not done:
            action = egreedy_policy(q_values, state, epsilon
            next_state, reward, done = env.step(action)
            reward_sum += reward
            q_values [state] [action] = q_values [state] [ac
            state = next_state

        ep_rewards.append(reward_sum)

    return ep_rewards, q_values
```

Now, let's the run Q-learning

In [30]:

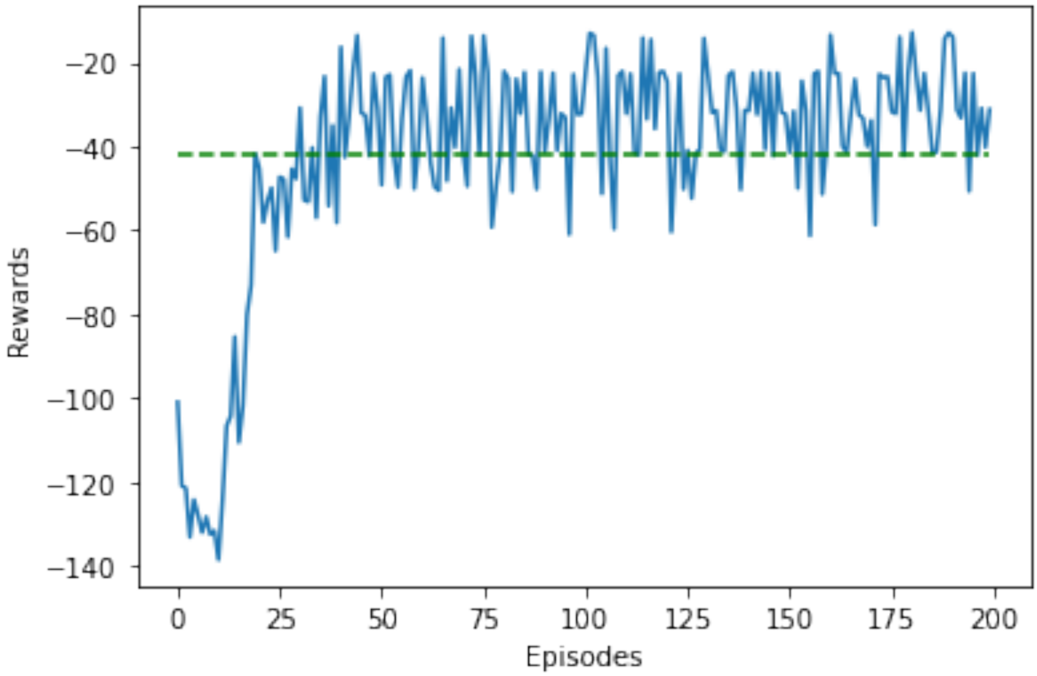
```
q_learning_rewards, q_values = q_learning(env, gamma=0.9, le

q_learning_rewards, _ = zip(*[q_learning(env, render=False,
avg_rewards = np.mean(q_learning_rewards, axis=0)
mean_reward = [np.mean(avg_rewards)] * len(avg_rewards)

fig, ax = plt.subplots()
ax.set_xlabel('Episodes')
ax.set_ylabel('Rewards')
ax.plot(avg_rewards)
ax.plot(mean_reward, 'g--')

print('Mean Reward: {}'.format(mean_reward[0]))
```

```
Mean Reward: -41.8545
```



Visualization

Finally, let's look at the policy learned

In [31]:

```
def play(q_values):
    env = GridWorld()
    state = env.reset()
    done = False

    while not done:
        # Select action
        action = egreedy_policy(q_values, state, 0.0)
        # Do the action
        next_state, reward, done = env.step(action)

        # Update state and action
        state = next_state

        env.render(q_values=q_values, action=action, coloriz
```

In [32]:

```
%matplotlib
play(q_values)
```

```
Using matplotlib backend: MacOSX
```

Q2 Constraint Satisfaction [Total: 10 points]

Written CS problem

Suppose we want to schedule some final exams for CS courses with the following course numbers: 1007, 3137, 3157, 3203, 3261, 4115, 4118, 4156

Suppose also that there are no students in common taking the following pairs of courses:

1007-3137

1007-3157, 3137-3157

1007-3203

1007-3261, 3137-3261, 3203-3261

1007-4115, 3137-4115, 3203-4115, 3261-4115

1007-4118, 3137-4118

1007-4156, 3137-4156, 3157-4156

How many exam slots are necessary to schedule exams?

There is a total of 8 courses, and we know that each course has a final exam. There will be a total of 8 final exams that need to be scheduled. Below I have attached the course list with the constraints applied of the courses in common that students have:

green = students take these course in common red = students do not take these courses in common

	1007	3137	3157	3203	3261	4115	4118	4156
1007	N/A							
3137		N/A						
3157			N/A					
3203				N/A				
3261					N/A			
4115						N/A		
4118							N/A	
4156								N/A

I will approach this scheduling problem with the following constraints:

- courses not taken in common can have their exams taken in the same day
- courses taken in common have to be scheduled on different days.

I will now go through the courses in order with the constraints applied:

- 1007 is not taken in common with any of the other courses so we know this course can have their final schedule with any other course.
- 3137 is not taken in common with 1007 so we can schedule them for the same day.
- 3157 is not taken in common with 1007 or 3137 so it can also be scheduled on the same day.
- 3203 is not taken in common with 1007 but it is taken in common with 3137 and 3157 so it needs to be scheduled for a separate day.
- 3261 is not taken in common with 1007 or 3137 but it is taken in common with 3157 so it can't be scheduled on that day; however, it is not taken in common with 3203 so it can be scheduled on the same day.
- 4115 is not taken in common with 1007 or 3137 but it is taken in common with 3157 so it can't be scheduled on that day. It is not taken in common with 3203 or 3261so it can be scheduled on the same day.
- 4118 is not taken in common with 1007 or 3137 but it is taken in common with every other course. It has to be scheduled on a separate day.
- 4156 is not taken in common with 1007, 3137, or 3157 so it can be scheduled on the same day as their exam.

In conclusion, 3 exam slots are necessary to schedule the exams for these courses with their constraints.

Days:	01	02	03
1007	T		
3137	T		
3157	T		
3203		T	
3261		T	
4115		T	
4118			T
4156	T		

In [ ]:

Processing math: 0%