Homework Exploration - report

Reinforcement Learning MVA 2024

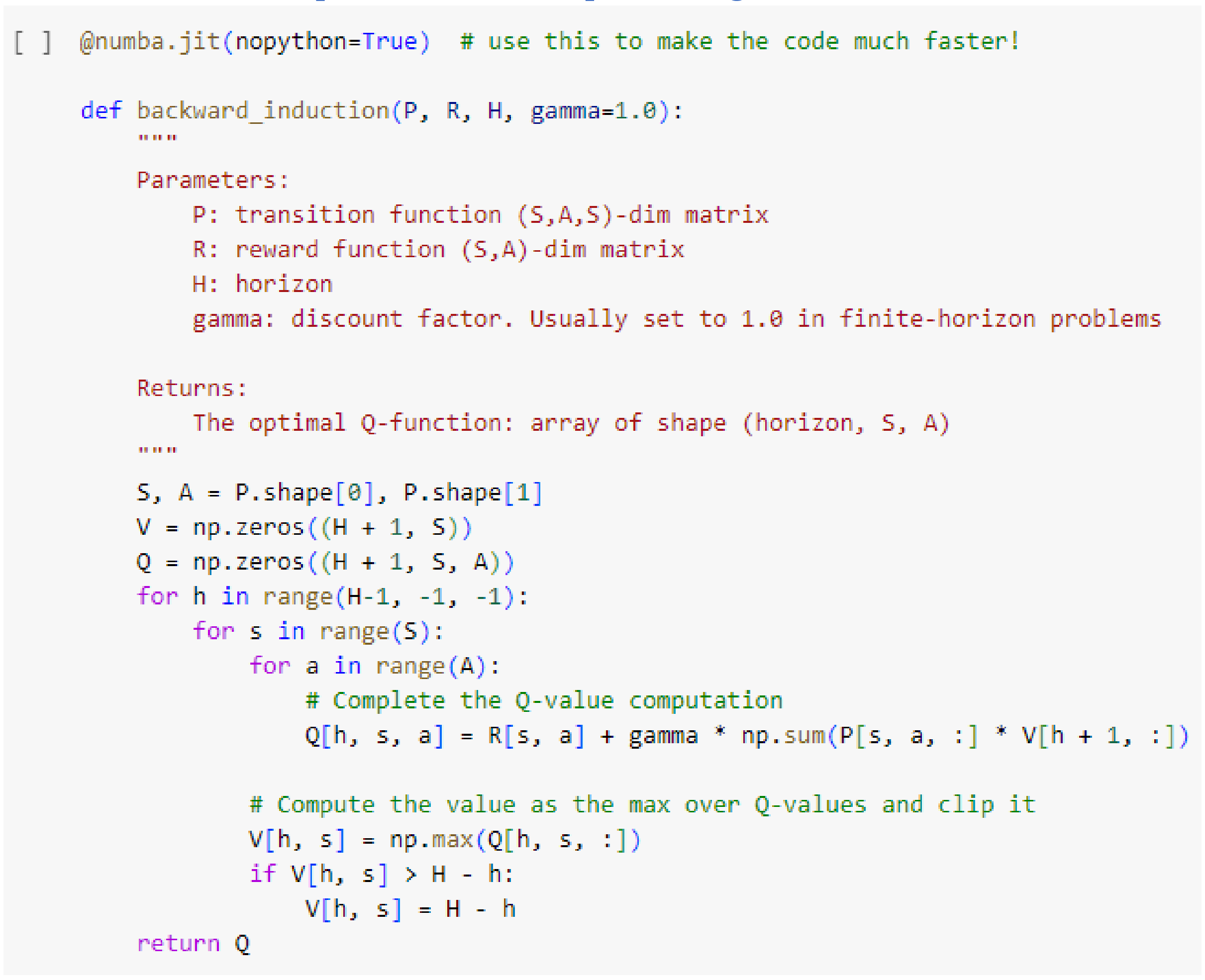
Student: David Faget Caño

Teacher: Emilie Kaufmann

The complete Colab notebook can be found at https://colab.research.google.com/drive/1SlP-sBe3AQwIGBM7KmmcZPCWrHXdSwFy?usp=sharing

Implementation of backward induction

The first cell to be completed is the one implementing value iteration. Here is the code:

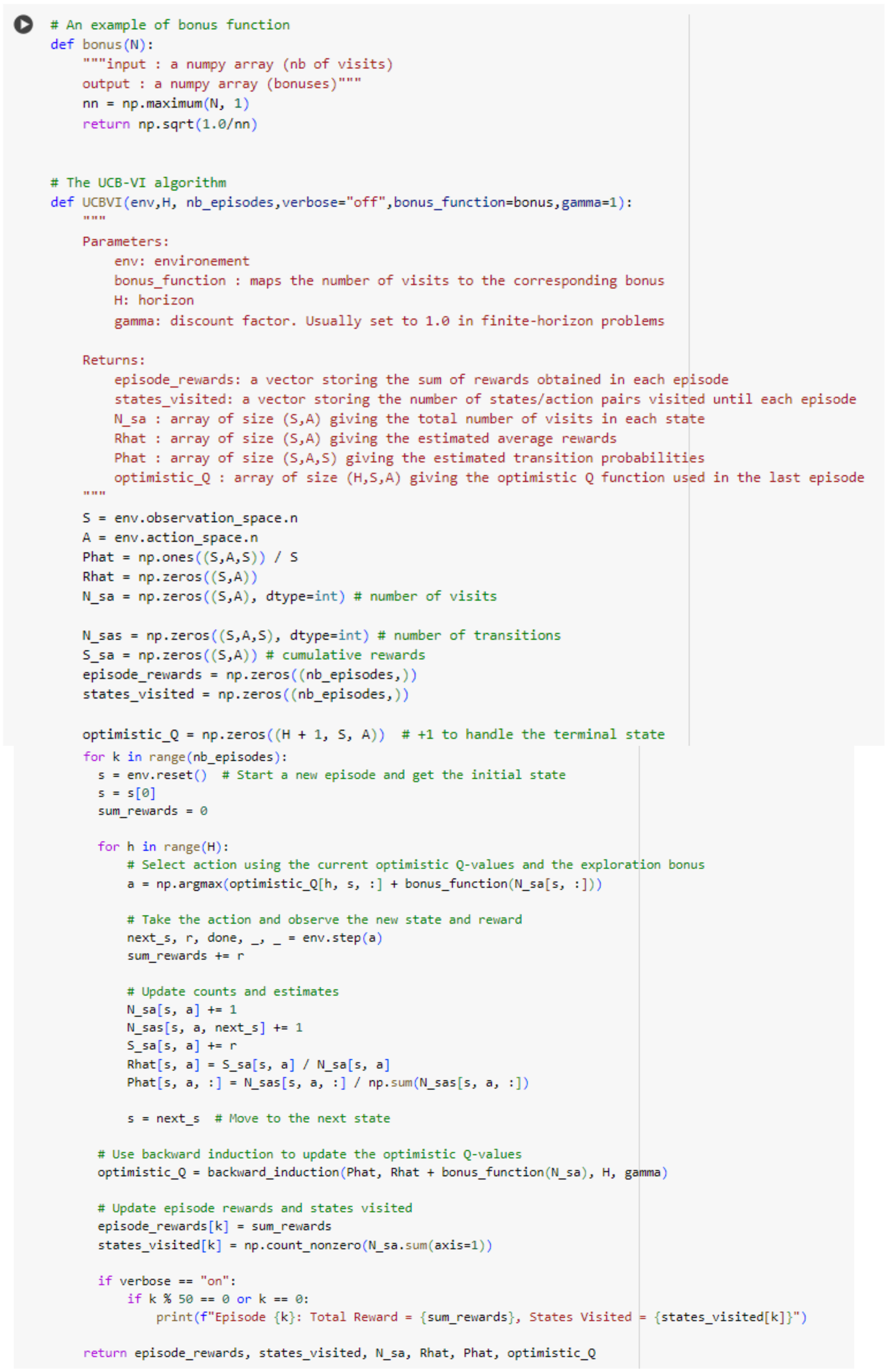


```python
@numba.jit(nopython=True)  # use this to make the code much faster!

def backward_induction(P, R, H, gamma=1.0):
    """
    Parameters:
        P: transition function (S,A,S)-dim matrix
        R: reward function (S,A)-dim matrix
        H: horizon
        gamma: discount factor. Usually set to 1.0 in finite-horizon problems

    Returns:
        The optimal Q-function: array of shape (horizon, S, A)
    """
    S, A = P.shape[0], P.shape[1]
    V = np.zeros((H + 1, S))
    Q = np.zeros((H + 1, S, A))
    for h in range(H-1, -1, -1):
        for s in range(S):
            for a in range(A):
                # Complete the Q-value computation
                Q[h, s, a] = R[s, a] + gamma * np.sum(P[s, a, :] * V[h + 1, :])

            # Compute the value as the max over Q-values and clip it
            V[h, s] = np.max(Q[h, s, :])
            if V[h, s] > H - h:
                V[h, s] = H - h
    return Q
```

In the notebook, we try it on the Gridworld environment to demonstrate that it works correctly.

Implementation of UCB-VI

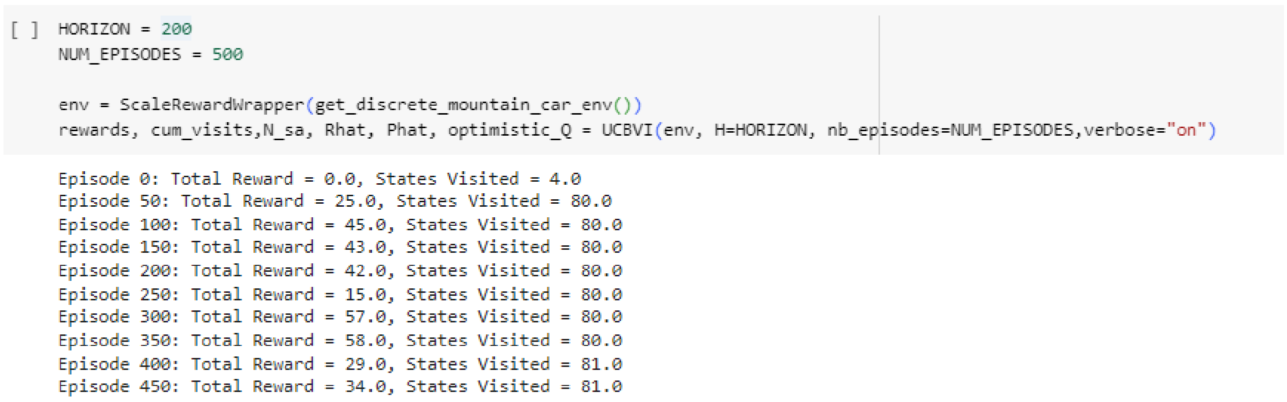
The code below successfully implements UCB-VI:



```python
# An example of bonus function
def bonus(N):
    """input : a numpy array (nb of visits)
    output : a numpy array (bonuses)"""
    nn = np.maximum(N, 1)
    return np.sqrt(1.0/nn)


# The UCB-VI algorithm
def UCBVI(env,H, nb_episodes,verbose="off",bonus_function=bonus,gamma=1):
    """
    Parameters:
        env: environement
        bonus_function : maps the number of visits to the corresponding bonus
        H: horizon
        gamma: discount factor. Usually set to 1.0 in finite-horizon problems

    Returns:
        episode_rewards: a vector storing the sum of rewards obtained in each episode
        states_visited: a vector storing the number of states/action pairs visited until each episode
        N_sa : array of size (S,A) giving the total number of visits in each state
        Rhat : array of size (S,A) giving the estimated average rewards
        Phat : array of size (S,A,S) giving the estimated transition probabilities
        optimistic_Q : array of size (H,S,A) giving the optimistic Q function used in the last episode
    """
    S = env.observation_space.n
    A = env.action_space.n
    Phat = np.ones((S,A,S)) / S
    Rhat = np.zeros((S,A))
    N_sa = np.zeros((S,A), dtype=int) # number of visits

    N_sas = np.zeros((S,A,S), dtype=int) # number of transitions
    S_sa = np.zeros((S,A)) # cumulative rewards
    episode_rewards = np.zeros((nb_episodes,))
    states_visited = np.zeros((nb_episodes,))

    optimistic_Q = np.zeros((H + 1, S, A))  # +1 to handle the terminal state
    for k in range(nb_episodes):
      s = env.reset()  # Start a new episode and get the initial state
      s = s[0]
      sum_rewards = 0

      for h in range(H):
          # Select action using the current optimistic Q-values and the exploration bonus
          a = np.argmax(optimistic_Q[h, s, :] + bonus_function(N_sa[s, :]))

          # Take the action and observe the new state and reward
          next_s, r, done, _, _ = env.step(a)
          sum_rewards += r

          # Update counts and estimates
          N_sa[s, a] += 1
          N_sas[s, a, next_s] += 1
          S_sa[s, a] += r
          Rhat[s, a] = S_sa[s, a] / N_sa[s, a]
          Phat[s, a, :] = N_sas[s, a, :] / np.sum(N_sas[s, a, :])

          s = next_s  # Move to the next state

      # Use backward induction to update the optimistic Q-values
      optimistic_Q = backward_induction(Phat, Rhat + bonus_function(N_sa), H, gamma)

      # Update episode rewards and states visited
      episode_rewards[k] = sum_rewards
      states_visited[k] = np.count_nonzero(N_sa.sum(axis=1))

      if verbose == "on":
          if k % 50 == 0 or k == 0:
              print(f"Episode {k}: Total Reward = {sum_rewards}, States Visited = {states_visited[k]}")

    return episode_rewards, states_visited, N_sa, Rhat, Phat, optimistic_Q
```

1. **To check whether the algorithm is working, you can visualize the amount of rewards gathered in episodes with the default bonus, as well as the number of visited states since the beginning (which measures how well the environment is being explored). Based on your findings, how many episodes seem necessary for the algorithm to behave well?**

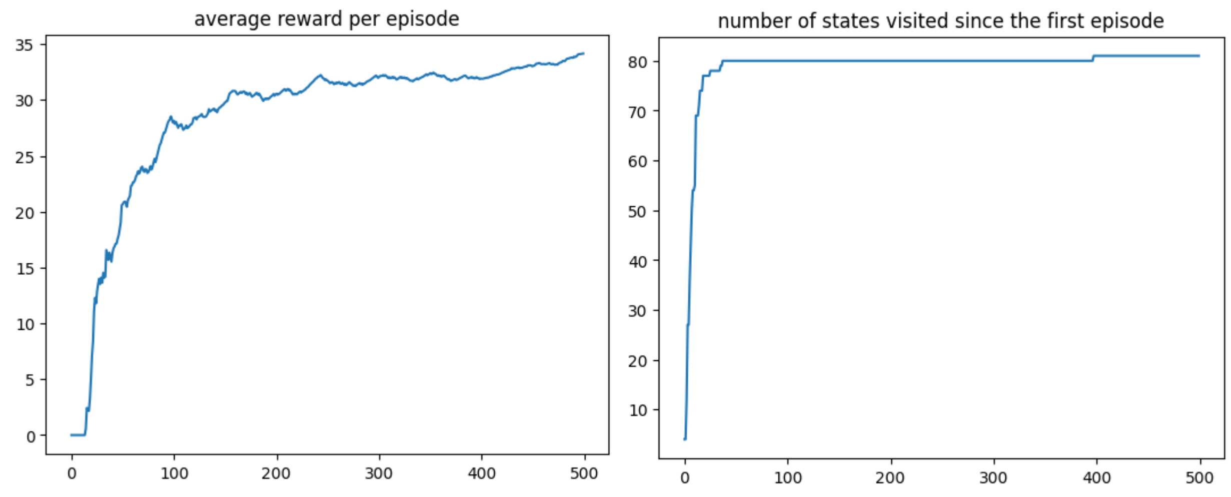
In order to answer this question, we may run the following cell:



```
[ ] HORIZON = 200
    NUM_EPISODES = 500

    env = ScaleRewardWrapper(get_discrete_mountain_car_env())
    rewards, cum_visits,N_sa, Rhat, Phat, optimistic_Q = UCBVI(env, H=HORIZON, nb_episodes=NUM_EPISODES,verbose="on")
```

```
Episode 0: Total Reward = 0.0, States Visited = 4.0
Episode 50: Total Reward = 25.0, States Visited = 80.0
Episode 100: Total Reward = 45.0, States Visited = 80.0
Episode 150: Total Reward = 43.0, States Visited = 80.0
Episode 200: Total Reward = 42.0, States Visited = 80.0
Episode 250: Total Reward = 15.0, States Visited = 80.0
Episode 300: Total Reward = 57.0, States Visited = 80.0
Episode 350: Total Reward = 58.0, States Visited = 80.0
Episode 400: Total Reward = 29.0, States Visited = 81.0
Episode 450: Total Reward = 34.0, States Visited = 81.0
```

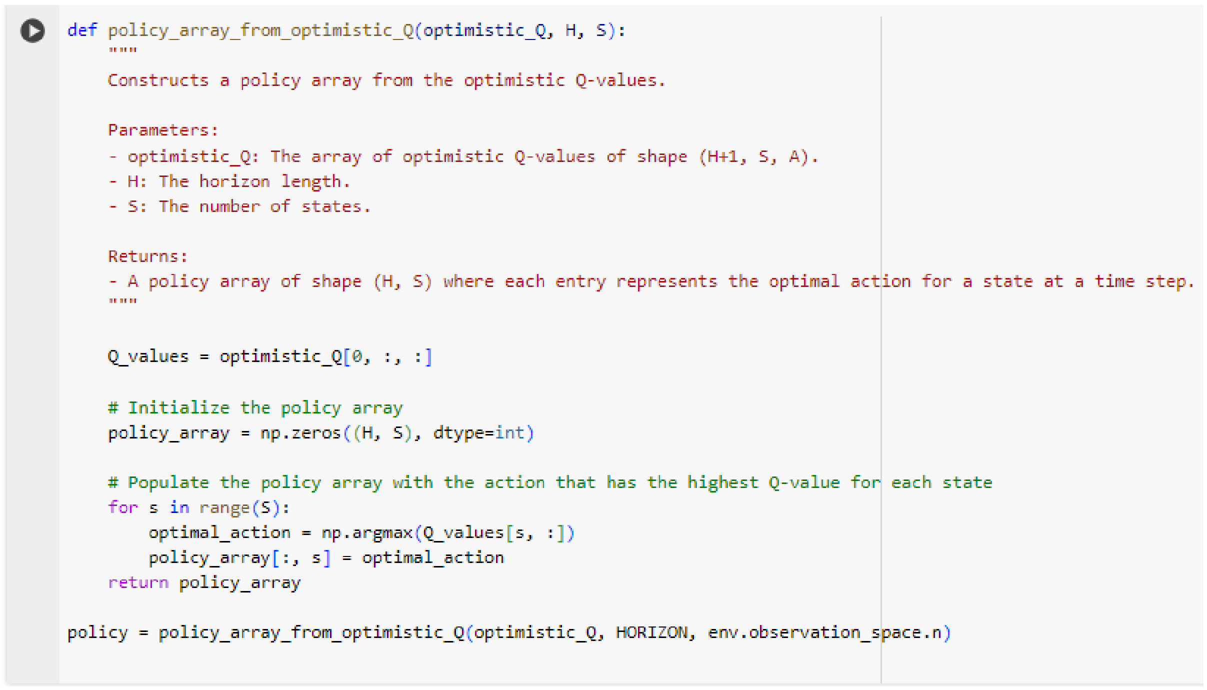
We can also plot the average reward per episode as well as the number of states visited since the first episode:



We observe that the environment is well explored pretty soon (after only 50 episodes). Initially, the agent explores a limited number of states in the early episodes. However, by the 50th episode, it consistently explores approximately 80 states per episode, indicating efficient and extensive exploration of the environment from that point onward. Yet, achieving a high average reward demands significantly more episodes. Based on observations from the initial plot, it appears that at least 300 episodes are required for the agent to attain satisfactory performance levels.

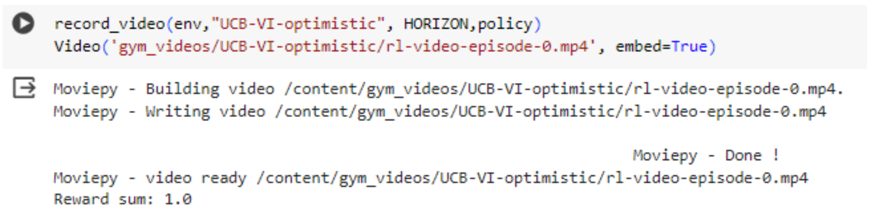
2. UCBVI's first purpose is not to output a candidate optimal policy, but if you want to do so, what is (are) reasonable candidate(s)? You can check by looking how the agent behaves under this (these) policies, i.e. whether it solves the task. You can also display a 2D visualization of associated Q functions.

We first have to choose the policy to visualize. We can employ the following function:



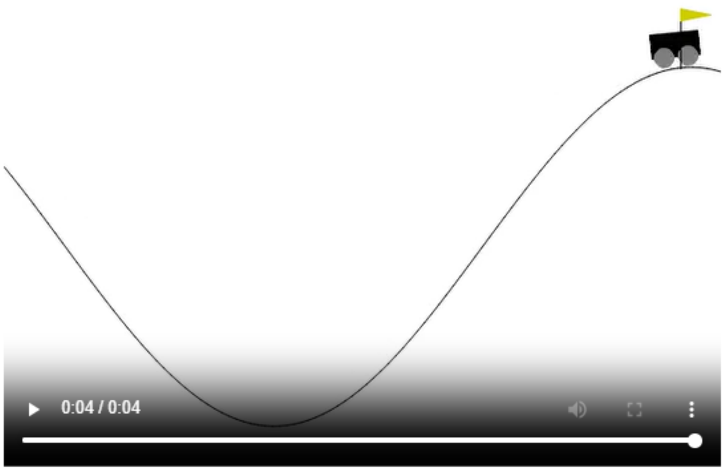
```python
def policy_array_from_optimistic_Q(optimistic_Q, H, S):
    """
    Constructs a policy array from the optimistic Q-values.

    Parameters:
    - optimistic_Q: The array of optimistic Q-values of shape (H+1, S, A).
    - H: The horizon length.
    - S: The number of states.

    Returns:
    - A policy array of shape (H, S) where each entry represents the optimal action for a state at a time step.
    """

    Q_values = optimistic_Q[0, :, :]

    # Initialize the policy array
    policy_array = np.zeros((H, S), dtype=int)

    # Populate the policy array with the action that has the highest Q-value for each state
    for s in range(S):
        optimal_action = np.argmax(Q_values[s, :])
        policy_array[:, s] = optimal_action
    return policy_array

policy = policy_array_from_optimistic_Q(optimistic_Q, HORIZON, env.observation_space.n)
```

In the notebook, we observe that the car successfully reaches de top of the hill:



```python
record_video(env,"UCB-VI-optimistic", HORIZON,policy)
Video('gym_videos/UCB-VI-optimistic/rl-video-episode-0.mp4', embed=True)
```

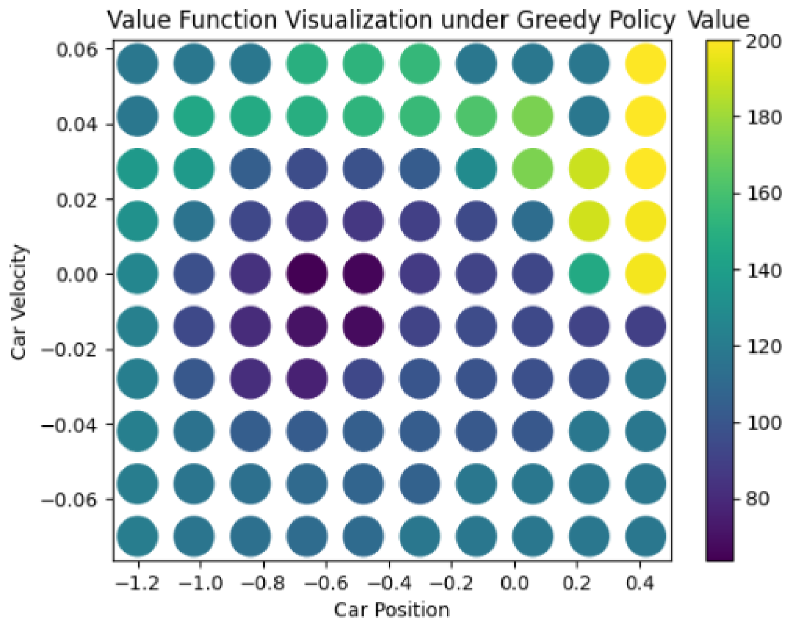
```
Moviepy - Building video /content/gym_videos/UCB-VI-optimistic/rl-video-episode-0.mp4.
Moviepy - Writing video /content/gym_videos/UCB-VI-optimistic/rl-video-episode-0.mp4

                                                              Moviepy - Done !
Moviepy - video ready /content/gym_videos/UCB-VI-optimistic/rl-video-episode-0.mp4
Reward sum: 1.0
```



Let's now display a 2D visualization of the value function under greedy policy:

```
[ ]  Q_function = optimistic_Q[0, :, :]
     # Compute the value function as the max over Q-values for each action in each state
     value = np.max(Q_function, axis=1)

     # Now, plot the value for each state.
     plt.scatter(env.discretized_states[0, :], env.discretized_states[1, :], c=value, s=400, cmap='viridis')
     plt.xlabel('Car Position')
     plt.ylabel('Car Velocity')
     clb = plt.colorbar()
     clb.ax.set_title('Value')
     plt.title('Value Function Visualization under Greedy Policy')
     plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/gymnasium/core.py:311: UserWarning: WARN: env.discretized_states i
  logger.warn(
```



We observe that this type of visualization is very useful in understanding what the agent has learned and how it perceives different states of the environment in terms of their potential for gaining reward. In particular, we see that states where the car has a position close to the goal (near 0.5) and a high velocity (towards the right, positive values) are considered to be of high value (colored yellow). This is expected, as these states mean the car is likely to reach the goal soon.

3. [Theory] For a given T, give an upper bound on the probability $\mathbb{P}(V^\star - V^{\hat{\pi}_T} > \varepsilon)$ that depends on T, ε and B_T (you can use Markov's inequality). Deduce the order of magnitude of the number of episodes needed by UCB-VI to output a ε-optimal policy with probability 1−δ. Comment on the result.

Applying Markov inequality we directly obtain

$$P\left(V^\star - V^{\widehat{\pi_T}} > \varepsilon\right) = P(R_T > \varepsilon) \le \frac{E[R_T]}{\varepsilon} \le \frac{B_T}{\varepsilon}$$

If we now focus on UCB-VI, we can find in existing literature such as [1] the following reasoning. To achieve an ε-optimal policy with probability at least $1 - \delta$, we set:

$$\frac{B_T}{\epsilon} \leq \delta$$

Solving for T gives us the number of episodes needed by UCB-VI to ensure that the probability of the regret being larger than ε is less than δ.

$$T \geq f(\epsilon, \delta, B_T)$$

In the expression above, the specific form of $f(\epsilon, \delta, B_T)$ depends on the expression of B_T, which is derived from the algorithm's theoretical analysis. For UCB-VI, as seen in class and in [1], it's common in UCB-type algorithms for B_T to have a dependency on $\log(T)$ and other parameters like S, A, and H (which are states, actions and horizon, respectively).

Therefore, the order of magnitude for T to achieve a ε-optimal policy with high probability (at least 1 – δ) is often polynomial in $1/\epsilon$, $1/\delta$, S, A, and H, and logarithmic in T itself.

Comments:

The requirement that the number of episodes grows polynomially with $1/\epsilon$ and $1/\delta$ is typical for reinforcement learning algorithms that provide high-probability guarantees on the performance of the learned policy (as seen for example in https://ar5iv.labs.arxiv.org/html/2107.07410). The presence of $\log(T)$ in B_T reflects the algorithm's efficiency in balancing exploration and exploitation; over time, less exploration is needed as the algorithm becomes more confident in its estimates. It assures us that with enough episodes, the algorithm will, with high probability, find a policy that is close to optimal. This theoretical guarantee is a strong point of UCB-VI. However, in practice, the number of episodes required might still be large, especially for complex environments with large state or action spaces (sometimes more complex than Mountain Car), showing the importance of efficient exploration and the choice of the bonus function.

[1] M. G. Azar, I. Osband, and R. Munos, “Minimax regret bounds for reinforcement learning,” in International Conference on Machine Learning. PMLR, 2017, pp. 263–272.

4. UCB-VI has actually some guarantees in terms of regret, so you can try to optimize the algorithm for the setting it is designed for.

Explore different kind of bonusses for UCB-VI, in terms of resulting *expected* cumulative rewards. You may also compare bonus-based "directed" exploration to a model-based algorithm using instead ϵ-greedy exploration.

In the following cell, we implement different bonusses:

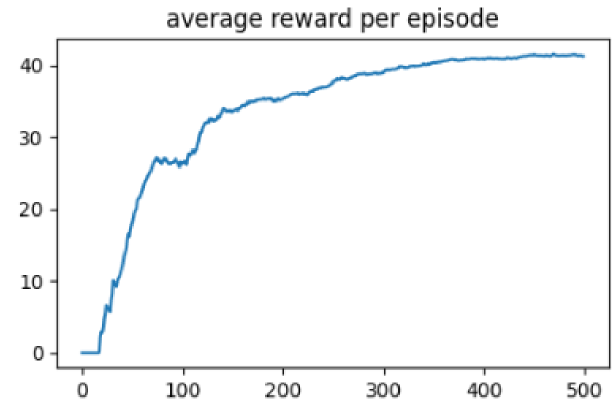
```
[ ] # First bonus but multiplied by 1/100

    def bonus_100(N):
        """input : a numpy array (nb of visits)
        output : a numpy array (bonuses)"""
        nn = np.maximum(N, 1)
        return (1/100)*np.sqrt(1.0/nn)


    # A logarithmic scaling can provide a slower decay of the bonus, potentially encouraging more exploration over a longer period:

    def logarithmic_bonus(N):
        nn = np.maximum(N, 1)
        return np.log(nn + 1) / nn

    # An inverse bonus provides a direct inverse relationship, which might lead to more aggressive exploration early on:

    def inverse_bonus(N):
        nn = np.maximum(N, 1)
        return 1.0 / nn

    # This bonus decreases exponentially with the number of visits, offering a fast-decreasing exploration incentive:

    def exponential_decay_bonus(N):
        return np.exp(-N)
```

Now we call UCB-VI for each bonus, setting the horizon to 200 and the number of episodes to 500.

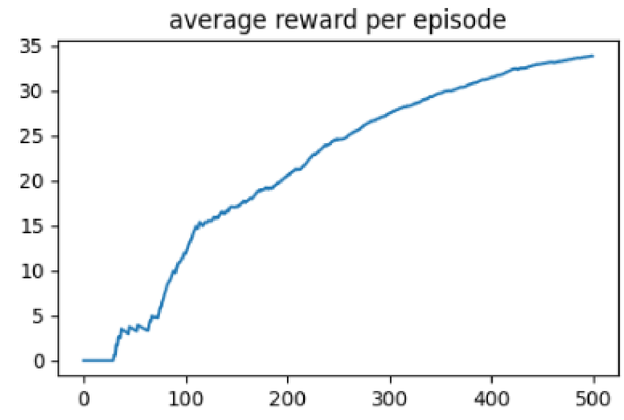
```
[ ] HORIZON = 200
    NUM_EPISODES = 500

    bonus_functions = {
        'bonus': bonus,
        'bonus_100': bonus_100,
        'logarithmic_bonus': logarithmic_bonus,
        'inverse_bonus': inverse_bonus,
        'exponential_decay_bonus': exponential_decay_bonus
    }


    for bonus_name in bonus_functions:
        env = ScaleRewardWrapper(get_discrete_mountain_car_env())
        # Retrieve the bonus function object by name
        bonus_func = bonus_functions[bonus_name]
        # Pass the bonus function object to UCBVI
        rewards, cum_visits, N_sa, Rhat, Phat, optimistic_Q = UCBVI(env, H=HORIZON, nb_episodes=NUM_EPISODES, bonus_function=bonus_func, verbose="off")
        print("________________________")
        print(f"Bonus name:{bonus_name}")
        print(f"Mean of last episode rewards:", np.mean(rewards))
        plt.figure(figsize=(5, 3))
        plt.plot([np.cumsum(rewards)[episode]/(episode+1) for episode in range(NUM_EPISODES)])
        plt.title("average reward per episode")
        plt.show()
```

We obtain the following results:

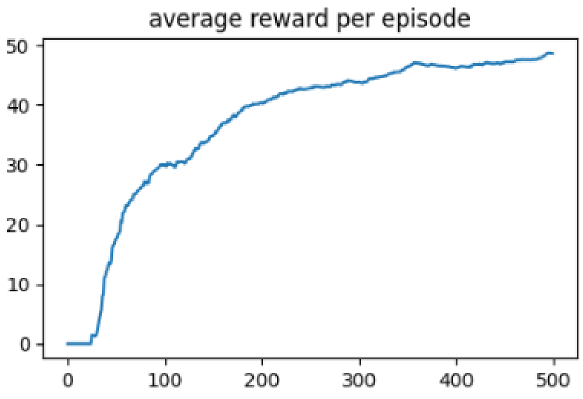
```
________________________
Bonus name:bonus
Mean of last episode rewards: 41.256
```



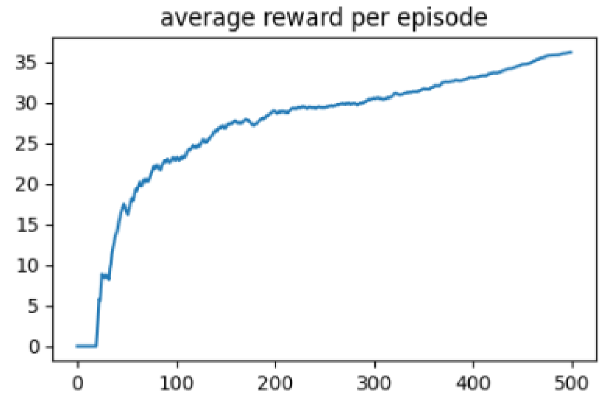
```
________________________
Bonus name:bonus_100
Mean of last episode rewards: 33.822
```



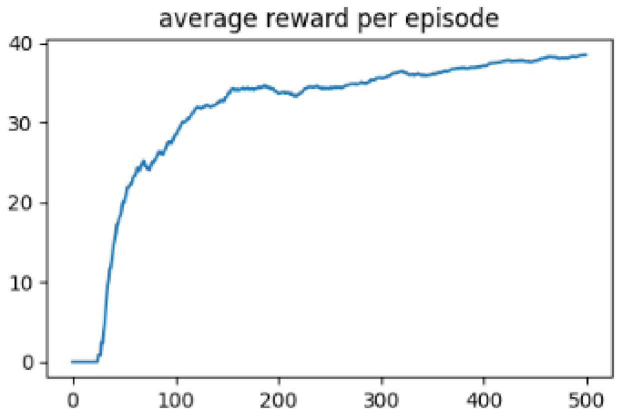
```
_________________________
Bonus name:logarithmic_bonus
Mean of last episode rewards: 48.622
```



```
_________________________
Bonus name:inverse_bonus
Mean of last episode rewards: 36.18
```

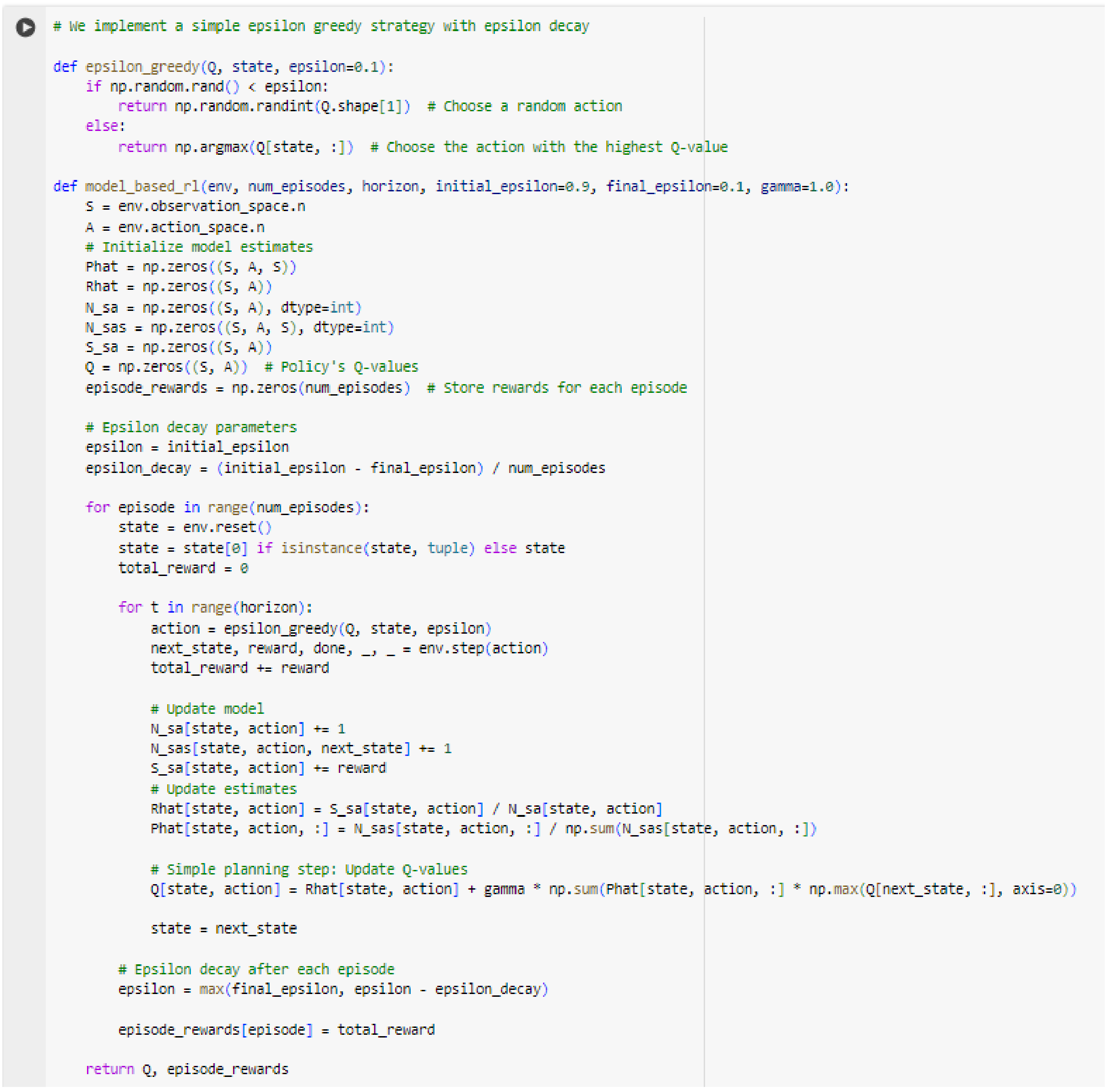


```
_________________________
Bonus name:exponential_decay_bonus
Mean of last episode rewards: 38.544
```



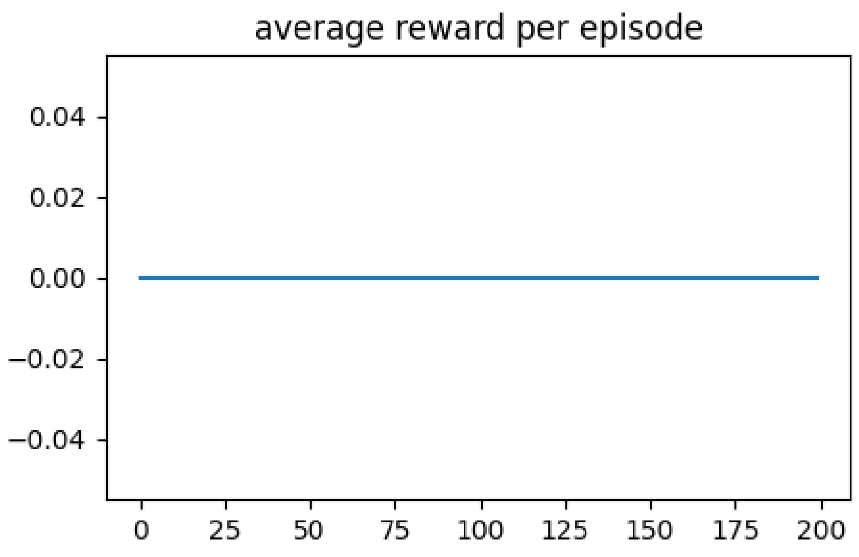
Comments: We observe that the bonus employed in the previous question ("bonus") and the logarithmic bonus seem to be the best choices. Yet, due to inherent randomness in execution, the performance in terms of rewards garnered can vary. At times, the straightforward "bonus" method might yield superior results, whereas in other instances, the "logarithmic_bonus" approach may prove to be more advantageous. Let's explore now a simple model-based algorithm which combines epsilon-greedy with epsilon decay.

Let's now compare this approach to a model based one, implementing a simple epsilon greedy strategy with epsilon decay. The following screenshot shows the implemented code:



```python
# We implement a simple epsilon greedy strategy with epsilon decay

def epsilon_greedy(Q, state, epsilon=0.1):
    if np.random.rand() < epsilon:
        return np.random.randint(Q.shape[1])  # Choose a random action
    else:
        return np.argmax(Q[state, :])  # Choose the action with the highest Q-value

def model_based_rl(env, num_episodes, horizon, initial_epsilon=0.9, final_epsilon=0.1, gamma=1.0):
    S = env.observation_space.n
    A = env.action_space.n
    # Initialize model estimates
    Phat = np.zeros((S, A, S))
    Rhat = np.zeros((S, A))
    N_sa = np.zeros((S, A), dtype=int)
    N_sas = np.zeros((S, A, S), dtype=int)
    S_sa = np.zeros((S, A))
    Q = np.zeros((S, A))  # Policy's Q-values
    episode_rewards = np.zeros(num_episodes)  # Store rewards for each episode

    # Epsilon decay parameters
    epsilon = initial_epsilon
    epsilon_decay = (initial_epsilon - final_epsilon) / num_episodes

    for episode in range(num_episodes):
        state = env.reset()
        state = state[0] if isinstance(state, tuple) else state
        total_reward = 0

        for t in range(horizon):
            action = epsilon_greedy(Q, state, epsilon)
            next_state, reward, done, _, _ = env.step(action)
            total_reward += reward

            # Update model
            N_sa[state, action] += 1
            N_sas[state, action, next_state] += 1
            S_sa[state, action] += reward
            # Update estimates
            Rhat[state, action] = S_sa[state, action] / N_sa[state, action]
            Phat[state, action, :] = N_sas[state, action, :] / np.sum(N_sas[state, action, :])

            # Simple planning step: Update Q-values
            Q[state, action] = Rhat[state, action] + gamma * np.sum(Phat[state, action, :] * np.max(Q[next_state, :], axis=0))

            state = next_state

        # Epsilon decay after each episode
        epsilon = max(final_epsilon, epsilon - epsilon_decay)

        episode_rewards[episode] = total_reward

    return Q, episode_rewards
```

We run it for 200 episodes with a horizon fixed to 5000. The following plot gives us the results obtained:



We observe that this simple strategy is completely inefficient. Indeed, the primary challenge in the Mountain Car environment is the sparsity of rewards. The agent only receives a reward when it reaches the top of the mountain. This sparsity makes it hard for the agent to learn effective policies, as it might not encounter positive rewards frequently enough to learn from them. An epsilon-greedy strategy can help explore the environment, but without a sophisticated approach to learning from sparse rewards, progress can be slow, as demonstrated by this example, where setting the horizon to 5000 is not enough.

5. [Theory] Our goal was to find a policy that manages to get the car up the hill. We chose to model this as solving an episodic MDP with a large enough horizon H. Could this task also be modelled as solving a discounted MDP? What value would you choose for the discount?

Yes, the task of finding a policy that successfully gets the car up the hill can also be modelled as solving a discounted MDP. As seen in class, in a discounted MDP, future rewards are multiplied by a discount factor γ (where $0 \leq \gamma < 1$) to reduce their present value. This approach helps in prioritizing immediate rewards over distant ones, which is particularly useful in continuous or very long tasks where an immediate action's effect might be more significant than those far in the future. We also saw in class that the discount factor γ significantly influences the agent's behaviour. A high γ, close to 1, encourages the agent to consider future rewards nearly as important as immediate ones, recommending long-term planning and strategic thinking. In contrast, a low γ makes the agent prioritize immediate rewards, which can result in short-sighted decisions that might not align with long-term objectives.

For the Mountain Car environment, we would likely choose a γ closer to 1. This is because the task requires a sequence of actions over multiple steps to achieve success, and the value of those actions may not be immediately apparent (since it is an environment with sparse rewards). A high γ ensures that the agent sufficiently values the future rewards associated with actions that contribute to the long-term goal of reaching the top of the hill. Choosing γ slightly less than 1 (for example, 0.95 or 0.99) seems a good approach for balancing the need to value future rewards while ensuring the sum of discounted rewards remains finite, even in infinite horizon scenarios.

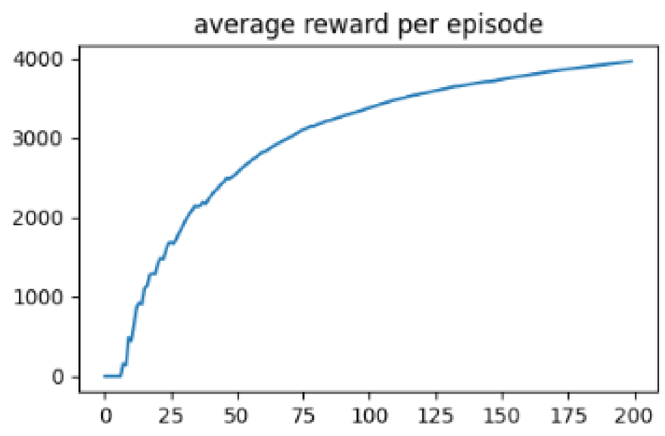
6. Compare the best model-based algorithm found in question 4. to a model-free alternative (still in terms of cumulative rewards).

We run the following code:

```
[ ]  import numpy as np

     def q_learning_with_exploration_bonus(env, num_episodes, horizon, initial_epsilon=0.9, final_epsilon=0.1, gamma=0.99, initial_q_value=1.0):
         S = env.observation_space.n
         A = env.action_space.n
         # Initialize Q-values optimistically
         Q = np.full((S, A), initial_q_value)
         N_sa = np.zeros((S, A), dtype=int)  # Number of visits to state-action pairs
         episode_rewards = np.zeros(num_episodes)  # Store rewards for each episode

         # Epsilon decay parameters
         epsilon = initial_epsilon
         epsilon_decay = (initial_epsilon - final_epsilon) / num_episodes

         for episode in range(num_episodes):
             state = env.reset()
             state = state[0] if isinstance(state, tuple) else state
             total_reward = 0

             for t in range(horizon):
                 action = epsilon_greedy(Q, state, epsilon)
                 next_state, reward, done, _, _ = env.step(action)
                 N_sa[state, action] += 1

                 # Compute exploration bonus (here we do not use the precedent function)
                 bonus = np.sqrt(1 / (N_sa[state, action] + 1))
                 adjusted_reward = reward + bonus  # Adjust reward with exploration bonus

                 # Q-Learning update
                 best_next_action = np.argmax(Q[next_state, :])
                 td_target = adjusted_reward + gamma * Q[next_state, best_next_action]
                 td_error = td_target - Q[state, action]
                 Q[state, action] += (1 / N_sa[state, action]) * td_error

                 total_reward += reward
                 state = next_state

             # Epsilon decay after each episode
             epsilon = max(final_epsilon, epsilon - epsilon_decay)

             episode_rewards[episode] = total_reward

         return Q, episode_rewards
```

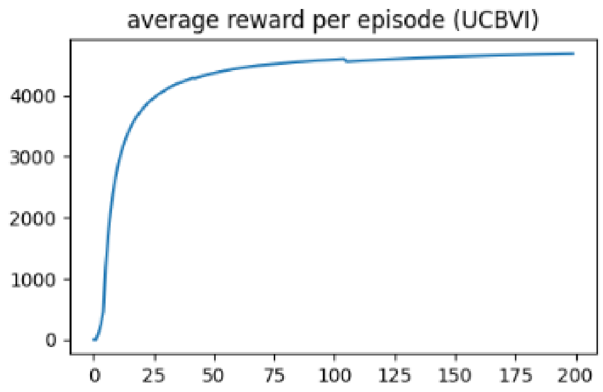
```
[ ]  HORIZON = 5000
     NUM_EPISODES = 200

     env = ScaleRewardWrapper(get_discrete_mountain_car_env())
     Q, rewards = q_learning_with_exploration_bonus(env, num_episodes=NUM_EPISODES, horizon = HORIZON)
     print("Mean of last episode rewards for epsilon-greedy:", np.mean(rewards))
     plt.figure(figsize=(5, 3))
     plt.plot([np.cumsum(rewards)[episode]/(episode+1) for episode in range(NUM_EPISODES)])
     plt.title("average reward per episode (model free)")
     plt.show()
```

```
Mean of last episode rewards for epsilon-greedy: 3961.1
```



```
[ ] # We compare it to UCBVI with the first bonus (same horizon and episodes)

    HORIZON = 5000
    NUM_EPISODES = 200

    env = ScaleRewardWrapper(get_discrete_mountain_car_env())
    rewards, cum_visits,N_sa, Rhat, Phat, optimistic_Q = UCBVI(env, H=HORIZON, nb_episodes=NUM_EPISODES,verbose="on")
    print("Mean of last episode rewards for epsilon-greedy:", np.mean(rewards))
    plt.figure(figsize=(5, 3))
    plt.plot([np.cumsum(rewards)[episode]/(episode+1) for episode in range(NUM_EPISODES)])
    plt.title("average reward per episode (UCBVI)")
    plt.show()
```

```
Episode 0: Total Reward = 0.0, States Visited = 49.0
Episode 50: Total Reward = 4826.0, States Visited = 80.0
Episode 100: Total Reward = 4774.0, States Visited = 80.0
Episode 150: Total Reward = 4848.0, States Visited = 80.0
Mean of last episode rewards for epsilon-greedy: 4689.985
```



Description of the chosen algorithm:

The algorithm chosen is an enhanced Q-learning algorithm designed for model-free reinforcement learning, which means it learns optimal actions by directly interacting with the environment without needing a model of the environment's dynamics (which differs from the model-based approach we used before). We are going to explain the model-free algorithm chosen:

1. **Initialization**: It starts with all Q-values (estimates of the expected rewards for taking certain actions in certain states) optimistically set high to encourage exploration. The number of visits to each state-action pair (N_sa) is initialized to zero.

2. **Exploration vs. Exploitation:** Actions are chosen using an epsilon-greedy strategy (function defined before in Question 4), which randomly chooses between exploring a new action or exploiting known information based on the epsilon value. Epsilon starts high for more exploration and gradually decreases to favour exploitation, which is coherent with our Mountain Car environment.

3. **Exploration Bonus**: As before, to encourage visiting less-explored state-action pairs, an exploration bonus is added to the reward, encouraging the agent to take "rare" actions. This bonus (calculated inline for simplicity rather than calling the function) decreases as a state-action pair gets visited more, promoting a balance between exploring new options and exploiting known strategies.

4. **Learning with Q-Updates**: After each action, the Q-value for the chosen state-action pair is updated based on the received reward (adjusted for

exploration), the best estimated future rewards, and the learning rate, which is adjusted to become smaller as a state-action pair is visited more often. This helps in making better the estimates of expected rewards over time.

5. **Epsilon Decay**: As time progresses, the value of epsilon gradually diminishes, changing the strategy from exploration of the environment towards exploitation of acquired knowledge for optimal results.

Comparison results:

Note that we have decided to utilize the first bonus proposed, since it gives better results than the logarithmic bonus in many runs of the notebook. The main observation is that in this case, even if both algorithms work and are able to learn a good policy, the model-free approach chosen takes more time to learn than UCBVI (and also accumulates less rewards after 200 episodes with Horizon = 5000).

7. [Theory] We would now like to compare the sample complexity of the model-free algorithm to that of the model-based algorithm. How would you proceed to do that?

In order to compare the sample complexity of model-free and model-based algorithms, we have to assess how many samples (or interactions with the environment) each type of algorithm requires to achieve a certain level of performance or accuracy in its policy or value function estimation. We would follow the following steps:

1. **Define Metrics**: We first have to choose how to measure performance (for example, average return per episode, or cumulative rewards like in this project) and sample complexity (for example, number of interactions with the environment to reach the goal).
2. **Tune Parameters**: We may ensure that both algorithms have optimally tuned hyperparameters for fairness.
3. **Run Experiments**: Then, we would have to train both algorithms over a significant number of episodes and evaluate their policies to obtain average performance metrics.
4. **Record and Compare**: Then, we would track the number of samples each algorithm needs to achieve similar performance levels.
5. **Analyse**: We could plot performance against sample count for both algorithms and analyse differences, considering exploration efficiency, model learning/updating overhead, and generalization capabilities.
6. **Conclude**: Finally, we would summarize the findings, emphasizing when one approach is more sample-efficient than the other and under what conditions.