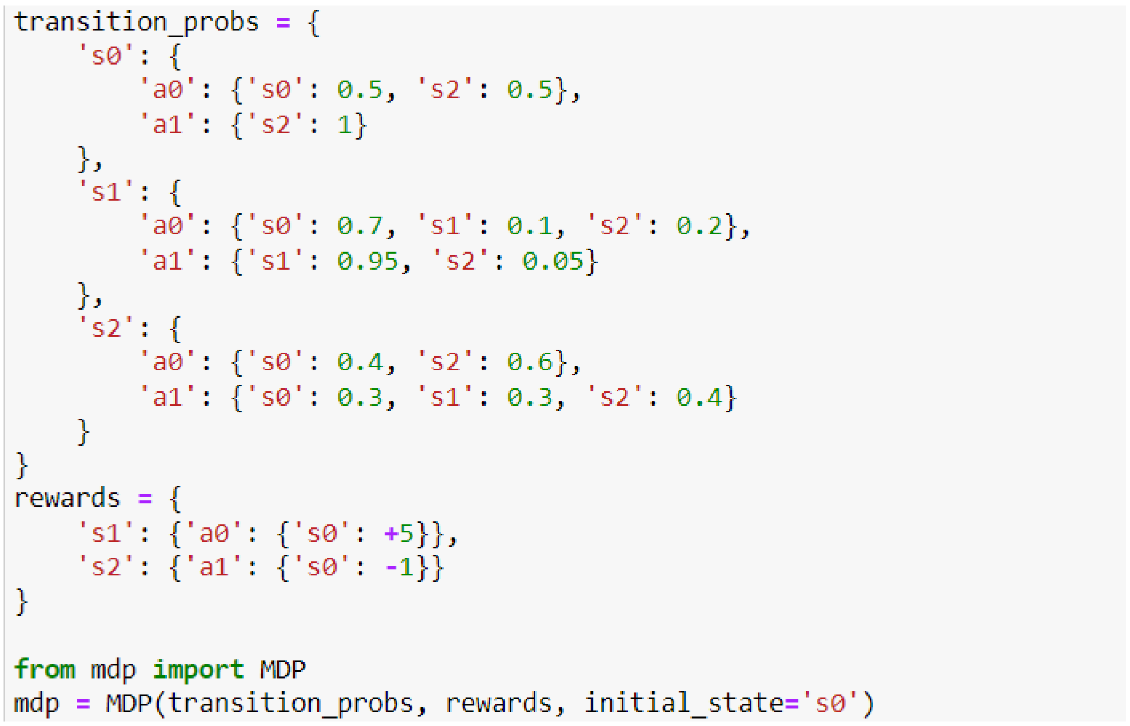
SIT796 Reinforcement Learning

Distinction Task 3.1D: Exact policy iteration implementation for MDPs

Markov Decision Process, which is a mathematical framework used to model sequential decision-making problems in reinforcement learning. In the context of the Mountain Car environment, the MDP represents the dynamics of the problem. It consists of a set of states, actions, transition probabilities, and rewards. The states represent different configurations of the car, such as position and velocity. The actions represent the agent's choices, such as applying force to the car in a certain direction.

In this task we are employing a linear system to find the exact solution for a policy in the Mountain Car environment and implementing a value iteration algorithm, which is an iterative method for solving Markov Decision Processes (MDPs).

The code defines the MDP using transition probabilities and rewards. The transition_probs dictionary represents the probabilities of transitioning from one state to another given an action, while the rewards dictionary defines the rewards associated with transitioning between states for specific actions.



```python
transition_probs = {
    's0': {
        'a0': {'s0': 0.5, 's2': 0.5},
        'a1': {'s2': 1}
    },
    's1': {
        'a0': {'s0': 0.7, 's1': 0.1, 's2': 0.2},
        'a1': {'s1': 0.95, 's2': 0.05}
    },
    's2': {
        'a0': {'s0': 0.4, 's2': 0.6},
        'a1': {'s0': 0.3, 's1': 0.3, 's2': 0.4}
    }
}
rewards = {
    's1': {'a0': {'s0': +5}},
    's2': {'a1': {'s0': -1}}
}

from mdp import MDP
mdp = MDP(transition_probs, rewards, initial_state='s0')
```

The value iteration algorithm is implemented using a loop that iteratively updates the state values until convergence. Inside the loop, the get_new_state_value function is called for each state to compute the next state value based on the current state values, transition probabilities, rewards, and discount factor (gamma). The value iteration loop updates the state values based on the new state values until the values converge. The get_action_value function computes the action value by summing up the expected immediate reward and the discounted future state values.

```python
# parameters
gamma = 0.9            # discount for MDP
num_iter = 1000         # maximum iterations, excluding initialization
# stop VI if new values are this close to old values (or closer)
min_difference = 0.0001

# initialize V(s)
state_values = {s: 0 for s in mdp.get_all_states()}

if has_graphviz:
    display(plot_graph_with_state_values(mdp, state_values))

for i in range(num_iter):
    # Compute new state values using the functions defined above.
    # It must be a dict {state : float V_new(state)}
    new_state_values = {}
    for s in mdp.get_all_states():
        nsv = get_new_state_value(mdp, state_values, s, gamma)
        a = list(nsv)[0]
        v = nsv[a]
        new_state_values[s] = v

    assert isinstance(new_state_values, dict)

    # Compute difference
    diff = max(abs(new_state_values[s] - state_values[s]) for s in mdp.get_all_states())
    print("iter %4i   |   diff: %6.5f   |   " % (i, diff), end="")
    print('   '.join("V(%s) = %.3f" % (s, v) for s, v in state_values.items()))
    state_values = new_state_values

    if diff < min_difference:
        print("Terminated")
        break
```

The underlying principle of value iteration is to solve a system of linear equations known as the Bellman equations, which can be represented as a linear system. The Bellman equations express the value of each state in terms of the values of its neighbouring states, taking into account the transition probabilities and rewards of the MDP. The concept of the Bellman equations is applied in the get_new_state_value and get_action_value functions. These functions compute the state values and action values, respectively, based on the Bellman equations.

```python
def get_action_value(mdp, state_values, state, action, gamma):
    # Initialize Q
    Q = 0
    for s in mdp.get_all_states():
        # Compute Q using the equation above
        Q += mdp.get_transition_prob(state, action, s) * (mdp.get_reward(state, action, s) +
                                                          gamma * state_values[s])
    return Q
```

```python
def get_new_state_value(mdp, state_values, state, gamma):
    # Computes next V(s) as in the formula above. Do not change state_values in the process.
    if mdp.is_terminal(state):
        return 0

    # Initialize the dict
    A = [a for a in mdp.get_possible_actions(state)]
    v = np.zeros(len(mdp.get_possible_actions(state)))
    i = 0

    # Compute all possible options
    for a in mdp.get_possible_actions(state):
        v[i] = get_action_value(mdp, state_values, state, a, gamma)
        A[i] = a
        i = i + 1

    # Recover V(s) and π*(s) as per the formula above
    V = {A[np.argmax(v)]: v[np.argmax(v)]}

    return V
```

By iteratively updating the state values based on the Bellman equations, the algorithm eventually converges to the optimal state values, which represent the expected return from each state under the optimal policy. The optimal policy can then be determined by choosing the action that maximizes the action value for each state. The loop continues until the maximum number of iterations is reached or the difference between the new state values and the current state values falls below a predefined threshold (min_difference).

After the value iteration loop converges and the state values have been computed, the code uses the get_optimal_action function to determine the optimal action for each state based on the computed state values and the Bellman equation.

```python
def get_optimal_action(mdp, state_values, state, gamma):
    # Finds optimal action using formula above.
    if mdp.is_terminal(state):
        return None

    nsv = get_new_state_value(mdp, state_values, state, gamma)
    a = max(nsv, key=nsv.get)

    return a
```

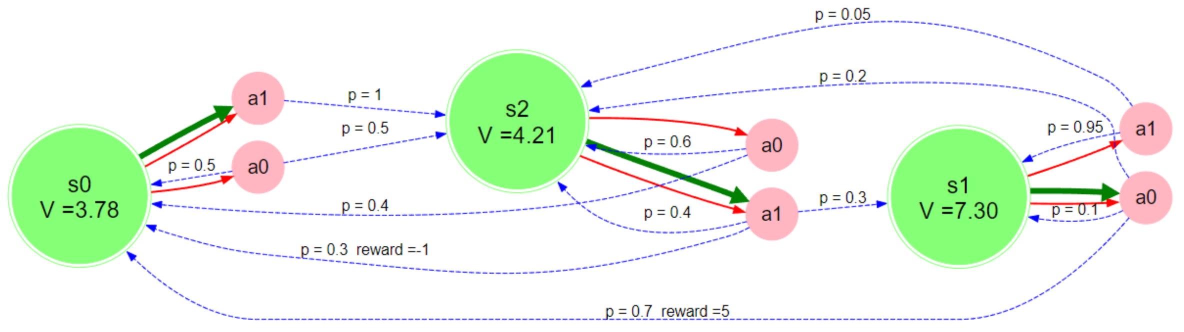
Lastly, the code simulates the environment using the optimal policy to calculate the average reward over 1000 steps. It initializes the environment, performs actions based on the optimal policy, and collects the rewards. The average reward is then computed using the np.mean function. Here, an average reward of 0.459 indicates that, on average, the agent is receiving positive rewards throughout its interactions with the Mountain Car environment. This suggests that the agent is making progress towards achieving its goal of maximizing its cumulative reward over time.

```
s = mdp.reset()
rewards = []
gamma = 0.9

for _ in range(1000):
    s, r, done, _ = mdp.step(get_optimal_action(mdp, state_values, s, gamma))
    rewards.append(r)

average_reward = np.mean(rewards)
print("average reward:", average_reward)

assert 0.40 < average_reward < 0.55
```

```
average reward: 0.459
```

By using Graphviz graph visualization, we can easily understand the optimal strategy and see the state values associated with each state. It provides a visual representation of how the policy can achieve the goal in the Mountain Car environment. The plot_graph_optimal_strategy_and_state_values function takes three arguments: the MDP object (mdp), the computed state values (state_values), and the get_action_value function. This function generates a graph where each state is represented as a node, and the edges between nodes indicate the transitions based on the optimal strategy. The state values are displayed as labels on the nodes.

```
if has_graphviz:
    display(plot_graph_optimal_strategy_and_state_values(mdp, state_values, get_action_value))
```


p = 0.05
p = 0.2
a1
p = 1
s2
V =4.21
p = 0.5
p = 0.6
a0
p = 0.95
a1
s0
V =3.78
p = 0.5
a0
p = 0.4
p = 0.4
a1
p = 0.3
s1
V =7.30
a0
p = 0.1
p = 0.3 reward =-1
p = 0.7 reward =5