Reinforcement Learning

Assignment-1

Submitted by:
Tayyib Ul Hassan (369648)

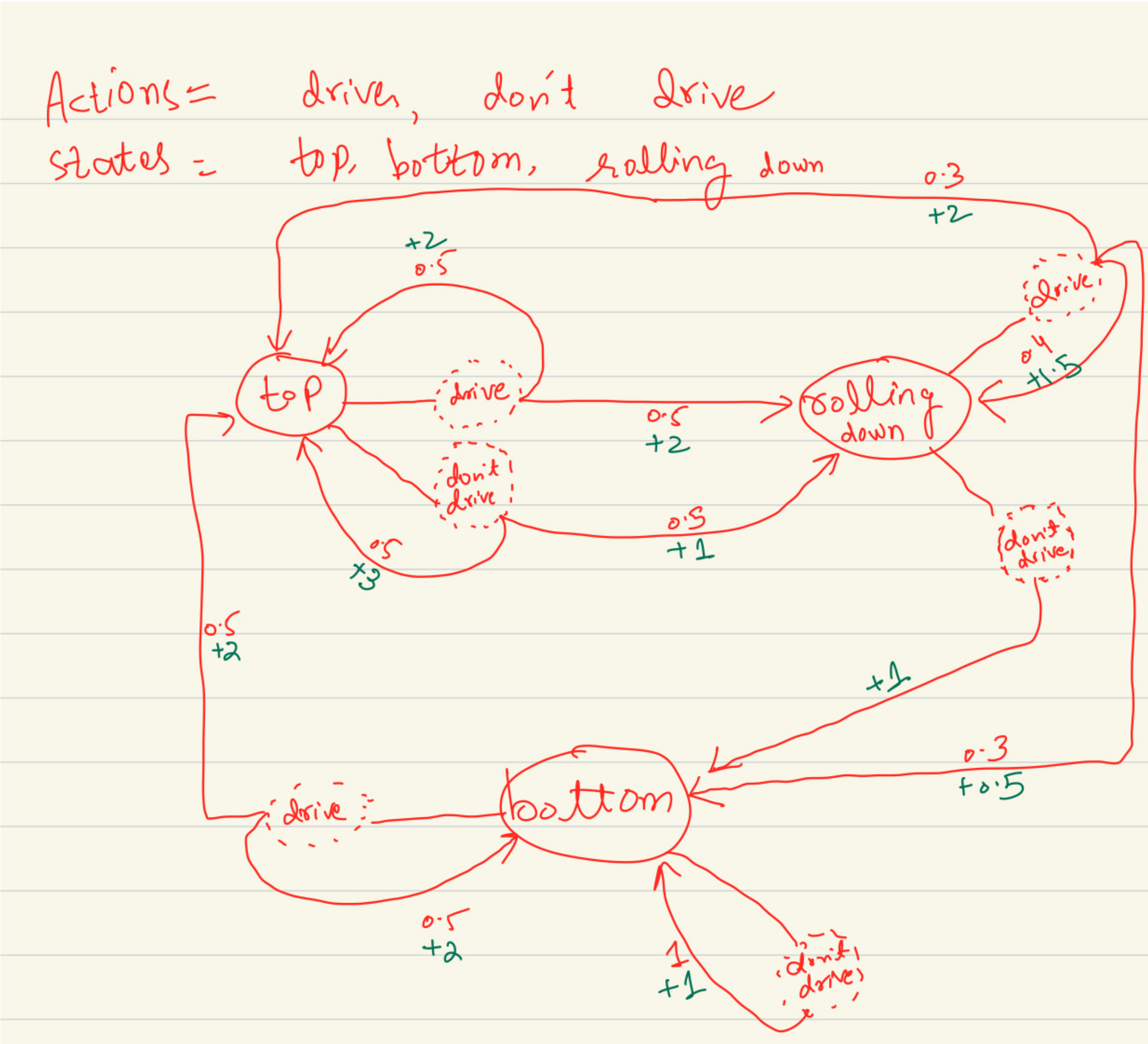
On:
27-10-2024

To:
Dr. Zuhair Zafar

School of Electrical engineering and Computer Sciences (SEECS),
NUST.

Question 1:

Below is the hand-drawn MDP of the problem specified in question statement.



Question 2:

- The question asked to implement value iteration on the given MDP.
- The discount factor which I used was 0.9.
- The values for all states are initially set to 0.

<u>Output Screenshot:</u>

```
(base) tayyibgondal@Tayyibs-MacBook-Air code-files % python value_iteration.py
Optimal value for state "top": 17.94
Policy for state "top": drive
----------------------------------------
Optimal value for state "rolling down": 17.48
Policy for state "rolling down": don't drive
----------------------------------------
Optimal value for state "bottom": 18.32
Policy for state "bottom": drive
----------------------------------------
```

Output Explanation:

The optimal policy suggests that, to maximize energy, the rover should **drive** when at the top or bottom of the hill but **not drive** while rolling down. This strategy balances the risks and rewards in each state, and achieves the highest long-term energy gain.

Code:

```python
# -------------------------------------------------
# imports
# -------------------------------------------------
import numpy as np

# -------------------------------------------------
# parameters
# -------------------------------------------------
num_states = 3 # (0=top, 1=rolling_down, 2=bottom)
num_actions = 2 # (0=drive, 1=no_drive)
discount_factor = 0.9

state_names = {0:'top', 1:'rolling down', 2:'bottom'}
action_names = {0:'drive', 1:"don't drive"}

# -------------------------------------------------
# initializing transition probabilities and rewards
# -------------------------------------------------
transitions = np.zeros((num_states, num_actions, num_states))
rewards = np.zeros((num_states, num_actions, num_states))

# -------------------------------------------------
# MDP Definition
# -------------------------------------------------
# Transition dynamics/uncertainties
transitions[0, 0, 0] = 0.5  # (top, drive, top)
transitions[0, 0, 1] = 0.5  # (top, drive, rolling_down)
transitions[0, 0, 2] = 0.0  # (top, drive, bottom)
transitions[0, 1, 0] = 0.5  # (top, no_drive, top)
transitions[0, 1, 1] = 0.5  # (top, no_drive, rolling_down)
transitions[0, 1, 2] = 0.0  # (top, no_drive, bottom)
transitions[1, 0, 0] = 0.3  # (rolling_down, drive, top)
```

```
transitions[1, 0, 1] = 0.4  # (rolling_down, drive, rolling_down)
transitions[1, 0, 2] = 0.3  # (rolling_down, drive, bottom)
transitions[1, 1, 0] = 0.0  # (rolling_down, no_drive, top)
transitions[1, 1, 1] = 0.0  # (rolling_down, no_drive, rolling_down)
transitions[1, 1, 2] = 1.0  # (rolling_down, no_drive, bottom)
transitions[2, 0, 0] = 0.5  # (bottom, drive, top)
transitions[2, 0, 1] = 0.0  # (bottom, drive, rolling_down)
transitions[2, 0, 2] = 0.5  # (bottom, drive, bottom)
transitions[2, 1, 0] = 0.0  # (bottom, no_drive, top)
transitions[2, 1, 1] = 0.0  # (bottom, no_drive, rolling_down)
transitions[2, 1, 2] = 1.0  # (bottom, no_drive, bottom)

# Rewards
rewards[0, 0, 0] = 2.0  # (top, drive, top)
rewards[0, 0, 1] = 2.0  # (top, drive, rolling_down)
rewards[0, 0, 2] = 0.0  # (top, drive, bottom)
rewards[0, 1, 0] = 3.0  # (top, no_drive, top)
rewards[0, 1, 1] = 1.0  # (top, no_drive, rolling_down)
rewards[0, 1, 2] = 0.0  # (top, no_drive, bottom)
rewards[1, 0, 0] = 2.0  # (rolling_down, drive, top)
rewards[1, 0, 1] = 1.5  # (rolling_down, drive, rolling_down)
rewards[1, 0, 2] = 0.5  # (rolling_down, drive, bottom)
rewards[1, 1, 0] = 0.0  # (rolling_down, no_drive, top)
rewards[1, 1, 1] = 0.0  # (rolling_down, no_drive, rolling_down)
rewards[1, 1, 2] = 1.0  # (rolling_down, no_drive, bottom)
rewards[2, 0, 0] = 2.0  # (bottom, drive, top)
rewards[2, 0, 1] = 0.0  # (bottom, drive, rolling_down)
rewards[2, 0, 2] = 2.0  # (bottom, drive, bottom)
rewards[2, 1, 0] = 0.0  # (bottom, no_drive, top)
rewards[2, 1, 1] = 0.0  # (bottom, no_drive, rolling_down)
rewards[2, 1, 2] = 1.0  # (bottom, no_drive, bottom)

# -------------------------------------------------
# value iteration
# -------------------------------------------------
def get_optimal_policy(values, transitions, rewards, discount_factor):
    '''
    once the value iteration is converved, this function is used to get
    the optimal policy based on converged v(s) values
    '''
    q_values = np.zeros((num_states, num_actions))

    # compute q(s, a) for all states and actions
    for s in range(num_states):
        for action in range(num_actions):
            q_values[s, action] = sum(transitions[s, action, s_next] * (rewards[s,
action, s_next] + discount_factor * values[s_next]) for s_next in range(num_states))
```

```
    # select the best actions as the policy for that state
    policy = np.argmax(q_values, axis=1)
    return policy


# value iteration
def value_iteration(transitions, rewards, discount_factor, tolerance=1e-6):
    values = np.zeros(num_states)

    while(True):
        delta = 0
        for s in range(num_states):
            v = values[s]

            # update value of current state from optimal v values of next states (from
last iteration)
            values[s] = max([sum(transitions[s, action, s_next] * (rewards[s, action,
s_next]+discount_factor*values[s_next]) for s_next in range(num_states)) for action in
range(num_actions)])

            delta = max(delta, abs(v-values[s]))

        # if values converge, break
        if delta < tolerance:
            break

    # once value iteration is converged, get the best policy
    policy = get_optimal_policy(values, transitions, rewards, discount_factor)

    return values, policy

# -------------------------------------------------
# Example usage - Value iteration
# -------------------------------------------------
values, policy = value_iteration(transitions, rewards, discount_factor)

for i in range(num_states):
    print(f'Optimal value for state "{state_names[i]}":', np.round(values[i], 2))
    print(f'Policy for state "{state_names[i]}":', action_names[policy[i]])
    print('---------------------------------------')
```

Question 3 (A): Policy iteration (starting with deterministic policy)

- The question asked to implement policy iteration, and test it by giving it deterministic policy at the start.
- Discount factor used = 0.9

- Initial policy was to ‘drive’ in all the states.

Console output after convergence:

```
(base) tayyibgondal@Tayyibs-MacBook-Air code-files % python policy_it
eration_with_deterministic_policy.py
Optimal value for state "top": 17.94
Policy for state "top": drive
---------------------------------------
Optimal value for state "rolling down": 17.48
Policy for state "rolling down": don't drive
---------------------------------------
Optimal value for state "bottom": 18.32
Policy for state "bottom": drive
---------------------------------------
```

Explanation:

Starting with a deterministic policy where the rover drives in all states, policy iteration gradually improved the policy based on state-value estimates, and converged to a policy where the rover should **drive when at the top or bottom** and **avoid driving while rolling down**. The same policy and the value estimates for states were observed with value iteration.

Code:

```python
# -------------------------------------------------
# imports
# -------------------------------------------------
import numpy as np

# -------------------------------------------------
# parameters
# -------------------------------------------------
num_states = 3 # (0=top, 1=rolling_down, 2=bottom)
num_actions = 2 # (0=drive, 1=no_drive)
discount_factor = 0.9

state_names = {0:'top', 1:'rolling down', 2:'bottom'}
action_names = {0:'drive', 1:"don't drive"}

# -------------------------------------------------
# initializing transition probabilities and rewards
# -------------------------------------------------
transitions = np.zeros((num_states, num_actions, num_states))
rewards = np.zeros((num_states, num_actions, num_states))

# -------------------------------------------------
# MDP Definition
# -------------------------------------------------
# Transition dynamics/uncertainties
transitions[0, 0, 0] = 0.5  # (top, drive, top)
```

```python
transitions[0, 0, 1] = 0.5  # (top, drive, rolling_down)
transitions[0, 0, 2] = 0.0  # (top, drive, bottom)
transitions[0, 1, 0] = 0.5  # (top, no_drive, top)
transitions[0, 1, 1] = 0.5  # (top, no_drive, rolling_down)
transitions[0, 1, 2] = 0.0  # (top, no_drive, bottom)
transitions[1, 0, 0] = 0.3  # (rolling_down, drive, top)
transitions[1, 0, 1] = 0.4  # (rolling_down, drive, rolling_down)
transitions[1, 0, 2] = 0.3  # (rolling_down, drive, bottom)
transitions[1, 1, 0] = 0.0  # (rolling_down, no_drive, top)
transitions[1, 1, 1] = 0.0  # (rolling_down, no_drive, rolling_down)
transitions[1, 1, 2] = 1.0  # (rolling_down, no_drive, bottom)
transitions[2, 0, 0] = 0.5  # (bottom, drive, top)
transitions[2, 0, 1] = 0.0  # (bottom, drive, rolling_down)
transitions[2, 0, 2] = 0.5  # (bottom, drive, bottom)
transitions[2, 1, 0] = 0.0  # (bottom, no_drive, top)
transitions[2, 1, 1] = 0.0  # (bottom, no_drive, rolling_down)
transitions[2, 1, 2] = 1.0  # (bottom, no_drive, bottom)

# Rewards
rewards[0, 0, 0] = 2.0  # (top, drive, top)
rewards[0, 0, 1] = 2.0  # (top, drive, rolling_down)
rewards[0, 0, 2] = 0.0  # (top, drive, bottom)
rewards[0, 1, 0] = 3.0  # (top, no_drive, top)
rewards[0, 1, 1] = 1.0  # (top, no_drive, rolling_down)
rewards[0, 1, 2] = 0.0  # (top, no_drive, bottom)
rewards[1, 0, 0] = 2.0  # (rolling_down, drive, top)
rewards[1, 0, 1] = 1.5  # (rolling_down, drive, rolling_down)
rewards[1, 0, 2] = 0.5  # (rolling_down, drive, bottom)
rewards[1, 1, 0] = 0.0  # (rolling_down, no_drive, top)
rewards[1, 1, 1] = 0.0  # (rolling_down, no_drive, rolling_down)
rewards[1, 1, 2] = 1.0  # (rolling_down, no_drive, bottom)
rewards[2, 0, 0] = 2.0  # (bottom, drive, top)
rewards[2, 0, 1] = 0.0  # (bottom, drive, rolling_down)
rewards[2, 0, 2] = 2.0  # (bottom, drive, bottom)
rewards[2, 1, 0] = 0.0  # (bottom, no_drive, top)
rewards[2, 1, 1] = 0.0  # (bottom, no_drive, rolling_down)
rewards[2, 1, 2] = 1.0  # (bottom, no_drive, bottom)

# ------------------------------------------------
# policy iteration
# ------------------------------------------------
def policy_evaluation(policy, transitions, rewards, discount_factor, tol=1e-6):
    values = np.zeros(num_states)
    # if all elements in policy are -1, then stochastic policy will be assumed
    stochastic = all(elem == -1 for elem in policy)

    while True:
        delta = 0
```

```
        for s in range(num_states):
            v = values[s]
            action = policy[s]

            # if policy is not stochastic
            if not stochastic:
                values[s] = sum(transitions[s, action, s_next] * (rewards[s, action,
s_next] + discount_factor * values[s_next]) for s_next in range(num_states))
            # if policy is stochastic, then average over all actions (equal
probability for each action)
            else:
                values[s] = sum(1/num_actions * sum(transitions[s, action, s_next] *
(rewards[s, action, s_next] + discount_factor * values[s_next]) for s_next in
range(num_states)) for action in range(num_actions))

            delta = max(delta, abs(v - values[s]))

        # if convergence achieved, then exit
        if delta < tol:
            break

    return values

def policy_iteration(policy, transitions, rewards, discount_factor, tol=1e-6):
    # Initialize q(s, a)
    q_values = np.zeros((num_states, num_actions))

    while True:
        stop = True  # Assume policy is stable initially (won't change, and we need to
exit after one policy iteration)
        # Evaluate the policy to get state values
        values = policy_evaluation(policy, transitions, rewards, discount_factor, tol)

        # Update the policy for each state
        for s in range(num_states):
            # Find the best action and its Q-value for this state
            best_action = policy[s]
            best_q_value = q_values[s, best_action]

            for action in range(num_actions):
                new_q_value = sum(
                    transitions[s, action, s_next] * (rewards[s, action, s_next] +
discount_factor * values[s_next])
                    for s_next in range(num_states)
                )

                # Update if this action is better than the best one found so far
                if new_q_value > best_q_value:
```

```
                    best_q_value = new_q_value
                    best_action = action
                    stop = False  # Mark that we made a policy change

            # Update policy and q_values with the best action for state s
            policy[s] = best_action
            q_values[s, best_action] = best_q_value

        # Stop if the policy is stable
        if stop:
            break

    return values, policy

# ------------------------------------------------
# Example usage - Deterministic policy
# ------------------------------------------------
policy = np.zeros(num_states, dtype=int)
policy[0] = 1
policy[1] = 1
policy[2] = 1

values, policy = policy_iteration(policy, transitions, rewards, discount_factor)

for i in range(num_states):
    print(f'Optimal value for state "{state_names[i]}":', np.round(values[i], 2))
    print(f'Policy for state "{state_names[i]}":', action_names[policy[i]])
    print('---------------------------------------')
```

Question 3 (B): Policy iteration (starting with stochastic policy)

- The question asked to implement policy iteration and test it by giving it deterministic policy at the start.
- Discount factor used = 0.9
- Initial policy was none, so the algorithm uses stochastic random (uniform) policy to start with. The code for this section is the same as above, except during calling the functions (the above code functions can work with either deterministic or stochastic policies).

Console output:

```
(base) tayyibgondal@Tayyibs-MacBook-Air code-files % python policy_iteration_with_stochastic_policy.py
Optimal value for state "top": 17.94
Policy for state "top": drive
---------------------------------------
Optimal value for state "rolling down": 17.48
Policy for state "rolling down": don't drive
---------------------------------------
Optimal value for state "bottom": 18.32
Policy for state "bottom": drive
---------------------------------------
```

Explanation:

For this task, I started with stochastic policy. Policy iteration again gradually improves the policy based on state-value estimates and converges to a policy where the rover should **drive when at the top or bottom** and **avoid driving while rolling down**. The same policy and the value estimates for states were observed with value iteration as well as with policy iteration (where we started with deterministic policy).

Code:

```
# -------------------------------------------------
# imports
# -------------------------------------------------
import numpy as np

# -------------------------------------------------
# parameters
# -------------------------------------------------
num_states = 3 # (0=top, 1=rolling_down, 2=bottom)
num_actions = 2 # (0=drive, 1=no_drive)
discount_factor = 0.9

state_names = {0:'top', 1:'rolling down', 2:'bottom'}
action_names = {0:'drive', 1:"don't drive"}

# -------------------------------------------------
# initializing transition probabilities and rewards
# -------------------------------------------------
transitions = np.zeros((num_states, num_actions, num_states))
rewards = np.zeros((num_states, num_actions, num_states))

# -------------------------------------------------
# MDP Definition
# -------------------------------------------------
# Transition dynamics/uncertainties
transitions[0, 0, 0] = 0.5  # (top, drive, top)
transitions[0, 0, 1] = 0.5  # (top, drive, rolling_down)
transitions[0, 0, 2] = 0.0  # (top, drive, bottom)
transitions[0, 1, 0] = 0.5  # (top, no_drive, top)
transitions[0, 1, 1] = 0.5  # (top, no_drive, rolling_down)
transitions[0, 1, 2] = 0.0  # (top, no_drive, bottom)
transitions[1, 0, 0] = 0.3  # (rolling_down, drive, top)
transitions[1, 0, 1] = 0.4  # (rolling_down, drive, rolling_down)
transitions[1, 0, 2] = 0.3  # (rolling_down, drive, bottom)
transitions[1, 1, 0] = 0.0  # (rolling_down, no_drive, top)
transitions[1, 1, 1] = 0.0  # (rolling_down, no_drive, rolling_down)
transitions[1, 1, 2] = 1.0  # (rolling_down, no_drive, bottom)
transitions[2, 0, 0] = 0.5  # (bottom, drive, top)
```

```
transitions[2, 0, 1] = 0.0  # (bottom, drive, rolling_down)
transitions[2, 0, 2] = 0.5  # (bottom, drive, bottom)
transitions[2, 1, 0] = 0.0  # (bottom, no_drive, top)
transitions[2, 1, 1] = 0.0  # (bottom, no_drive, rolling_down)
transitions[2, 1, 2] = 1.0  # (bottom, no_drive, bottom)

# Rewards
rewards[0, 0, 0] = 2.0  # (top, drive, top)
rewards[0, 0, 1] = 2.0  # (top, drive, rolling_down)
rewards[0, 0, 2] = 0.0  # (top, drive, bottom)
rewards[0, 1, 0] = 3.0  # (top, no_drive, top)
rewards[0, 1, 1] = 1.0  # (top, no_drive, rolling_down)
rewards[0, 1, 2] = 0.0  # (top, no_drive, bottom)
rewards[1, 0, 0] = 2.0  # (rolling_down, drive, top)
rewards[1, 0, 1] = 1.5  # (rolling_down, drive, rolling_down)
rewards[1, 0, 2] = 0.5  # (rolling_down, drive, bottom)
rewards[1, 1, 0] = 0.0  # (rolling_down, no_drive, top)
rewards[1, 1, 1] = 0.0  # (rolling_down, no_drive, rolling_down)
rewards[1, 1, 2] = 1.0  # (rolling_down, no_drive, bottom)
rewards[2, 0, 0] = 2.0  # (bottom, drive, top)
rewards[2, 0, 1] = 0.0  # (bottom, drive, rolling_down)
rewards[2, 0, 2] = 2.0  # (bottom, drive, bottom)
rewards[2, 1, 0] = 0.0  # (bottom, no_drive, top)
rewards[2, 1, 1] = 0.0  # (bottom, no_drive, rolling_down)
rewards[2, 1, 2] = 1.0  # (bottom, no_drive, bottom)

# -------------------------------------------------
# policy iteration
# -------------------------------------------------
def policy_evaluation(policy, transitions, rewards, discount_factor, tol=1e-6):
    values = np.zeros(num_states)
    # if all elements in policy are -1, then stochastic policy will be assumed
    stochastic = all(elem == -1 for elem in policy)

    while True:
        delta = 0
        for s in range(num_states):
            v = values[s]
            action = policy[s]

            # if policy is not stochastic
            if not stochastic:
                values[s] = sum(transitions[s, action, s_next] * (rewards[s, action,
s_next] + discount_factor * values[s_next]) for s_next in range(num_states))
            # if policy is stochastic, then average over all actions (equal
probability for each action)
            else:
```

```
                values[s] = sum(1/num_actions * sum(transitions[s, action, s_next] *
(rewards[s, action, s_next] + discount_factor * values[s_next]) for s_next in
range(num_states)) for action in range(num_actions))

            delta = max(delta, abs(v - values[s]))

        # if convergence achieved, then exit
        if delta < tol:
            break

    return values

def policy_iteration(policy, transitions, rewards, discount_factor, tol=1e-6):
    # Initialize q(s, a)
    q_values = np.zeros((num_states, num_actions))

    while True:
        stop = True  # Assume policy is stable initially (won't change, and we need to
exit after one policy iteration)
        # Evaluate the policy to get state values
        values = policy_evaluation(policy, transitions, rewards, discount_factor, tol)

        # Update the policy for each state
        for s in range(num_states):
            # Find the best action and its Q-value for this state
            best_action = policy[s]
            best_q_value = q_values[s, best_action]

            for action in range(num_actions):
                new_q_value = sum(
                    transitions[s, action, s_next] * (rewards[s, action, s_next] +
discount_factor * values[s_next])
                    for s_next in range(num_states)
                )

                # Update if this action is better than the best one found so far
                if new_q_value > best_q_value:
                    best_q_value = new_q_value
                    best_action = action
                    stop = False  # Mark that we made a policy change

            # Update policy and q_values with the best action for state s
            policy[s] = best_action
            q_values[s, best_action] = best_q_value

        # Stop if the policy is stable
        if stop:
            break
```

```python
    return values, policy

# -------------------------------------------------
# Example usage - Stochastic policy
# -------------------------------------------------
policy = np.zeros(num_states, dtype=int)
# stochastic policy
# (no policy is specified for any state, so policy iteration will use stochastic
policy at the start)
policy[0] = -1
policy[1] = -1
policy[2] = -1

values, policy = policy_iteration(policy, transitions, rewards, discount_factor)

for i in range(num_states):
    print(f'Optimal value for state "{state_names[i]}":', np.round(values[i], 2))
    print(f'Policy for state "{state_names[i]}":', action_names[policy[i]])
    print('---------------------------------------')
```

Question 4 (a): Value iteration with changed discount (0.75)

Changes made:

I reduced the discount factor, which affects how much future rewards matter. With a lower discount factor, the rover cares less about rewards it will get later and focuses more on immediate rewards.

Console screenshot:

```
(base) tayyibgondal@Tayyibs-MacBook-Air code-files % python value_iteration_with_changed_discount
.py
Optimal value for state "top": 7.19
Policy for state "top": don't drive
---------------------------------------
Optimal value for state "rolling down": 6.66
Policy for state "rolling down": drive
---------------------------------------
Optimal value for state "bottom": 7.52
Policy for state "bottom": drive
---------------------------------------
```

Explanation for changed output:

Reducing the discount factor decreases the importance of future rewards in the rover's decision-making process. This shift leads to a more short-sighted strategy, and the rover focuses on immediate gains rather than long-term benefits.

At the top, the rover opts **not to drive** because the immediate reward for staying still (3 units) outweighs the risk of driving, which could lead to rolling down with less certain rewards. The immediate benefit now takes precedence due to the reduced discounting of future rewards.

While Rolling Down**,** the rover chooses to **drive** because the immediate reward of moving down (1.5 units or 2 units) is more attractive than the potential risks associated with not driving and waiting to reach the bottom, especially since future rewards are discounted more heavily.

At the **Bottom** state, the rover's optimal policy remains to **drive** because of the immediate reward structure. When the rover drives from the bottom, it has a 50% chance of reaching the top (earning 2 units) and a 50% chance of staying at the bottom (earning 2 units).

With a reduced discount factor, future rewards are less valuable than before, so the rover focuses on the immediate outcomes of its actions. Driving from the bottom provides a better chance to gain energy quickly. Thus, the rover chooses to drive to maximize its immediate energy gain, which shows the impact of the lower discount factor on its decision-making.

Code:

```python
# ------------------------------------------------
# imports
# ------------------------------------------------
import numpy as np

# ------------------------------------------------
# parameters
# ------------------------------------------------
num_states = 3 # (0=top, 1=rolling_down, 2=bottom)
num_actions = 2 # (0=drive, 1=no_drive)

state_names = {0:'top', 1:'rolling down', 2:'bottom'}
action_names = {0:'drive', 1:"don't drive"}

# ------------------------------------------------
# initializing transition probabilities and rewards
# ------------------------------------------------
transitions = np.zeros((num_states, num_actions, num_states))
rewards = np.zeros((num_states, num_actions, num_states))

# ------------------------------------------------
# MDP Definition
# ------------------------------------------------
# Transition dynamics/uncertainties
transitions[0, 0, 0] = 0.5  # (top, drive, top)
transitions[0, 0, 1] = 0.5  # (top, drive, rolling_down)
transitions[0, 0, 2] = 0.0  # (top, drive, bottom)
transitions[0, 1, 0] = 0.5  # (top, no_drive, top)
```

```python
transitions[0, 1, 1] = 0.5  # (top, no_drive, rolling_down)
transitions[0, 1, 2] = 0.0  # (top, no_drive, bottom)
transitions[1, 0, 0] = 0.3  # (rolling_down, drive, top)
transitions[1, 0, 1] = 0.4  # (rolling_down, drive, rolling_down)
transitions[1, 0, 2] = 0.3  # (rolling_down, drive, bottom)
transitions[1, 1, 0] = 0.0  # (rolling_down, no_drive, top)
transitions[1, 1, 1] = 0.0  # (rolling_down, no_drive, rolling_down)
transitions[1, 1, 2] = 1.0  # (rolling_down, no_drive, bottom)
transitions[2, 0, 0] = 0.5  # (bottom, drive, top)
transitions[2, 0, 1] = 0.0  # (bottom, drive, rolling_down)
transitions[2, 0, 2] = 0.5  # (bottom, drive, bottom)
transitions[2, 1, 0] = 0.0  # (bottom, no_drive, top)
transitions[2, 1, 1] = 0.0  # (bottom, no_drive, rolling_down)
transitions[2, 1, 2] = 1.0  # (bottom, no_drive, bottom)

# Rewards
rewards[0, 0, 0] = 2.0  # (top, drive, top)
rewards[0, 0, 1] = 2.0  # (top, drive, rolling_down)
rewards[0, 0, 2] = 0.0  # (top, drive, bottom)
rewards[0, 1, 0] = 3.0  # (top, no_drive, top)
rewards[0, 1, 1] = 1.0  # (top, no_drive, rolling_down)
rewards[0, 1, 2] = 0.0  # (top, no_drive, bottom)
rewards[1, 0, 0] = 2.0  # (rolling_down, drive, top)
rewards[1, 0, 1] = 1.5  # (rolling_down, drive, rolling_down)
rewards[1, 0, 2] = 0.5  # (rolling_down, drive, bottom)
rewards[1, 1, 0] = 0.0  # (rolling_down, no_drive, top)
rewards[1, 1, 1] = 0.0  # (rolling_down, no_drive, rolling_down)
rewards[1, 1, 2] = 1.0  # (rolling_down, no_drive, bottom)
rewards[2, 0, 0] = 2.0  # (bottom, drive, top)
rewards[2, 0, 1] = 0.0  # (bottom, drive, rolling_down)
rewards[2, 0, 2] = 2.0  # (bottom, drive, bottom)
rewards[2, 1, 0] = 0.0  # (bottom, no_drive, top)
rewards[2, 1, 1] = 0.0  # (bottom, no_drive, rolling_down)
rewards[2, 1, 2] = 1.0  # (bottom, no_drive, bottom)

# ------------------------------------------------
# value iteration
# ------------------------------------------------
def get_optimal_policy(values, transitions, rewards, discount_factor):
    '''
    once the value iteration is converved, this function is used to get
    the optimal policy based on converged v(s) values
    '''
    q_values = np.zeros((num_states, num_actions))

    # compute q(s, a) for all states and actions
    for s in range(num_states):
        for action in range(num_actions):
```

```
            q_values[s, action] = sum(transitions[s, action, s_next] * (rewards[s,
action, s_next] + discount_factor * values[s_next]) for s_next in range(num_states))

    # select the best actions as the policy for that state
    policy = np.argmax(q_values, axis=1)
    return policy


# value iteration
def value_iteration(transitions, rewards, discount_factor, tolerance=1e-6):
    values = np.zeros(num_states)

    while(True):
        delta = 0
        for s in range(num_states):
            v = values[s]

            # update value of current state from optimal v values of next states (from
last iteration)
            values[s] = max([sum(transitions[s, action, s_next] * (rewards[s, action,
s_next]+discount_factor*values[s_next]) for s_next in range(num_states)) for action in
range(num_actions)])

            delta = max(delta, abs(v-values[s]))

        # if values converge, break
        if delta < tolerance:
            break

    # once value iteration is converged, get the best policy
    policy = get_optimal_policy(values, transitions, rewards, discount_factor)

    return values, policy

# -------------------------------------------------
# Example usage - Value iteration with changed discount
# -------------------------------------------------
discount_factor = 0.75

values, policy = value_iteration(transitions, rewards, discount_factor)

for i in range(num_states):
    print(f'Optimal value for state "{state_names[i]}":', np.round(values[i], 2))
    print(f'Policy for state "{state_names[i]}":', action_names[policy[i]])
    print('----------------------------------------')
```

Question 4 (b): Value iteration with changed transition probability for a particular state and action

Changes made:

I changed the chances of the rover moving when it drives from the **Rolling Down** state.

```
# changing probabilities
transitions[1, 0, 0] = 0.8  # (rolling_down, drive, top)           # old = 0.3
transitions[1, 0, 1] = 0.2  # (rolling_down, drive, rolling_down)  # old = 0.4
transitions[1, 0, 2] = 0.0  # (rolling_down, drive, bottom)        # old = 0.3
```

This means driving is more likely to help the rover get to the top, which is better for collecting energy.

Console screenshot:

```
(base) tayyibgondal@Tayyibs-MacBook-Air code-files % python value_iteration_with_changed_transiti
on_probabilities.py
Optimal value for state "top": 19.65
Policy for state "top": drive
---------------------------------------
Optimal value for state "rolling down": 19.57
Policy for state "rolling down": drive
---------------------------------------
Optimal value for state "bottom": 19.71
Policy for state "bottom": drive
---------------------------------------
```

Explanation for changed output:

The changes in transition probabilities made it much more likely for the rover to return to the **Top** state after driving from the **Rolling Down** state, now at **80%** chance. This increased chance means that driving from **Rolling Down** is now a better choice, leading to higher rewards when the rover gets back to the top. As a result, the optimal value for the **Rolling Down** state increased to **19.57**, and the best action changed to **drive**. The optimal values for the **Top** and **Bottom** states also went up to **19.65** and **19.71**, which shows that the rover's new strategy is better for collecting energy in all states.

Code:

```
# -------------------------------------------------------
# imports
# -------------------------------------------------------
import numpy as np

# -------------------------------------------------------
# parameters
# -------------------------------------------------------
num_states = 3 # (0=top, 1=rolling_down, 2=bottom)
num_actions = 2 # (0=drive, 1=no_drive)
```

```
discount_factor = 0.9

state_names = {0:'top', 1:'rolling down', 2:'bottom'}
action_names = {0:'drive', 1:"don't drive"}

# -------------------------------------------------
# initializing transition probabilities and rewards
# -------------------------------------------------
transitions = np.zeros((num_states, num_actions, num_states))
rewards = np.zeros((num_states, num_actions, num_states))

# -------------------------------------------------
# MDP Definition
# -------------------------------------------------
# Transition dynamics/uncertainties
transitions[0, 0, 0] = 0.5  # (top, drive, top)
transitions[0, 0, 1] = 0.5  # (top, drive, rolling_down)
transitions[0, 0, 2] = 0.0  # (top, drive, bottom)
transitions[0, 1, 0] = 0.5  # (top, no_drive, top)
transitions[0, 1, 1] = 0.5  # (top, no_drive, rolling_down)
transitions[0, 1, 2] = 0.0  # (top, no_drive, bottom)
transitions[1, 0, 0] = 0.3  # (rolling_down, drive, top)
transitions[1, 0, 1] = 0.4  # (rolling_down, drive, rolling_down)
transitions[1, 0, 2] = 0.3  # (rolling_down, drive, bottom)
transitions[1, 1, 0] = 0.0  # (rolling_down, no_drive, top)
transitions[1, 1, 1] = 0.0  # (rolling_down, no_drive, rolling_down)
transitions[1, 1, 2] = 1.0  # (rolling_down, no_drive, bottom)
transitions[2, 0, 0] = 0.5  # (bottom, drive, top)
transitions[2, 0, 1] = 0.0  # (bottom, drive, rolling_down)
transitions[2, 0, 2] = 0.5  # (bottom, drive, bottom)
transitions[2, 1, 0] = 0.0  # (bottom, no_drive, top)
transitions[2, 1, 1] = 0.0  # (bottom, no_drive, rolling_down)
transitions[2, 1, 2] = 1.0  # (bottom, no_drive, bottom)

# Rewards
rewards[0, 0, 0] = 2.0  # (top, drive, top)
rewards[0, 0, 1] = 2.0  # (top, drive, rolling_down)
rewards[0, 0, 2] = 0.0  # (top, drive, bottom)
rewards[0, 1, 0] = 3.0  # (top, no_drive, top)
rewards[0, 1, 1] = 1.0  # (top, no_drive, rolling_down)
rewards[0, 1, 2] = 0.0  # (top, no_drive, bottom)
rewards[1, 0, 0] = 2.0  # (rolling_down, drive, top)
rewards[1, 0, 1] = 1.5  # (rolling_down, drive, rolling_down)
rewards[1, 0, 2] = 0.5  # (rolling_down, drive, bottom)
rewards[1, 1, 0] = 0.0  # (rolling_down, no_drive, top)
rewards[1, 1, 1] = 0.0  # (rolling_down, no_drive, rolling_down)
rewards[1, 1, 2] = 1.0  # (rolling_down, no_drive, bottom)
rewards[2, 0, 0] = 2.0  # (bottom, drive, top)
```

```python
rewards[2, 0, 1] = 0.0  # (bottom, drive, rolling_down)
rewards[2, 0, 2] = 2.0  # (bottom, drive, bottom)
rewards[2, 1, 0] = 0.0  # (bottom, no_drive, top)
rewards[2, 1, 1] = 0.0  # (bottom, no_drive, rolling_down)
rewards[2, 1, 2] = 1.0  # (bottom, no_drive, bottom)

# -------------------------------------------------
# value iteration
# -------------------------------------------------
def get_optimal_policy(values, transitions, rewards, discount_factor):
    '''
    once the value iteration is converved, this function is used to get
    the optimal policy based on converged v(s) values
    '''
    q_values = np.zeros((num_states, num_actions))

    # compute q(s, a) for all states and actions
    for s in range(num_states):
        for action in range(num_actions):
            q_values[s, action] = sum(transitions[s, action, s_next] * (rewards[s,
action, s_next] + discount_factor * values[s_next]) for s_next in range(num_states))

    # select the best actions as the policy for that state
    policy = np.argmax(q_values, axis=1)
    return policy


# value iteration
def value_iteration(transitions, rewards, discount_factor, tolerance=1e-6):
    values = np.zeros(num_states)

    while(True):
        delta = 0
        for s in range(num_states):
            v = values[s]

            # update value of current state from optimal v values of next states (from
last iteration)
            values[s] = max([sum(transitions[s, action, s_next] * (rewards[s, action,
s_next]+discount_factor*values[s_next]) for s_next in range(num_states)) for action in
range(num_actions)])

            delta = max(delta, abs(v-values[s]))

        # if values converge, break
        if delta < tolerance:
            break
```

```python
    # once value iteration is converged, get the best policy
    policy = get_optimal_policy(values, transitions, rewards, discount_factor)

    return values, policy

# -------------------------------------------------
# Example usage - Value iteration with changed transition probabilities
# -------------------------------------------------
# changing probabilities
transitions[1, 0, 0] = 0.8  # (rolling_down, drive, top)
transitions[1, 0, 1] = 0.2  # (rolling_down, drive, rolling_down)
transitions[1, 0, 2] = 0.0  # (rolling_down, drive, bottom)

values, policy = value_iteration(transitions, rewards, discount_factor)

for i in range(num_states):
    print(f'Optimal value for state "{state_names[i]}":', np.round(values[i], 2))
    print(f'Policy for state "{state_names[i]}":', action_names[policy[i]])
    print('---------------------------------------')
```

Question 4 (c): Value iteration with changed rewards for particular state and action

Changes made:

I also changed the rewards for driving when the rover is **Rolling Down**. Now, if it drives and reaches the **Top**, it gets **4 points** instead of **2 points**. The rewards for staying in the **Rolling Down** state or going to the **Bottom** stay the same. This change makes driving a better choice because it gives the rover more points, encouraging it to drive more often.

```python
# changing rewards
rewards[1, 0, 0] = 4.0  # (rolling_down, drive, top)          # old reward = 2
rewards[1, 0, 1] = 1.5  # (rolling_down, drive, rolling_down)
rewards[1, 0, 2] = 0.5  # (rolling_down, drive, bottom)
```

Console screenshot:

```
(base) tayyibgondal@Tayyibs-MacBook-Air code-files % python value_iteration_with_changed_reward.p
y
Optimal value for state "top": 19.83
Policy for state "top": drive
---------------------------------------
Optimal value for state "rolling down": 19.79
Policy for state "rolling down": drive
---------------------------------------
Optimal value for state "bottom": 19.86
Policy for state "bottom": drive
---------------------------------------
```

Explanation for changed output:

With the old environment dynamics, the optimal action at rolling down state was chosen to be 'not drive' by the agent. However, when I increased the reward for the action of driving which led to the top state, from +2 to +4, the agent updated its new optimal policy. It now prefers 'driving' at the rolling down state to maximize the overall energy.
If we reduce this reward too much, the optimal action will again change to 'don't drive'.

Code:

```
# --------------------------------------------------
# imports
# --------------------------------------------------
import numpy as np

# --------------------------------------------------
# parameters
# --------------------------------------------------
num_states = 3 # (0=top, 1=rolling_down, 2=bottom)
num_actions = 2 # (0=drive, 1=no_drive)
discount_factor = 0.9

state_names = {0:'top', 1:'rolling down', 2:'bottom'}
action_names = {0:'drive', 1:"don't drive"}

# --------------------------------------------------
# initializing transition probabilities and rewards
# --------------------------------------------------
transitions = np.zeros((num_states, num_actions, num_states))
rewards = np.zeros((num_states, num_actions, num_states))

# --------------------------------------------------
# MDP Definition
# --------------------------------------------------
# Transition dynamics/uncertainties
transitions[0, 0, 0] = 0.5  # (top, drive, top)
transitions[0, 0, 1] = 0.5  # (top, drive, rolling_down)
transitions[0, 0, 2] = 0.0  # (top, drive, bottom)
transitions[0, 1, 0] = 0.5  # (top, no_drive, top)
transitions[0, 1, 1] = 0.5  # (top, no_drive, rolling_down)
transitions[0, 1, 2] = 0.0  # (top, no_drive, bottom)
transitions[1, 0, 0] = 0.3  # (rolling_down, drive, top)
transitions[1, 0, 1] = 0.4  # (rolling_down, drive, rolling_down)
transitions[1, 0, 2] = 0.3  # (rolling_down, drive, bottom)
transitions[1, 1, 0] = 0.0  # (rolling_down, no_drive, top)
transitions[1, 1, 1] = 0.0  # (rolling_down, no_drive, rolling_down)
transitions[1, 1, 2] = 1.0  # (rolling_down, no_drive, bottom)
```

```python
transitions[2, 0, 0] = 0.5  # (bottom, drive, top)
transitions[2, 0, 1] = 0.0  # (bottom, drive, rolling_down)
transitions[2, 0, 2] = 0.5  # (bottom, drive, bottom)
transitions[2, 1, 0] = 0.0  # (bottom, no_drive, top)
transitions[2, 1, 1] = 0.0  # (bottom, no_drive, rolling_down)
transitions[2, 1, 2] = 1.0  # (bottom, no_drive, bottom)

# Rewards
rewards[0, 0, 0] = 2.0  # (top, drive, top)
rewards[0, 0, 1] = 2.0  # (top, drive, rolling_down)
rewards[0, 0, 2] = 0.0  # (top, drive, bottom)
rewards[0, 1, 0] = 3.0  # (top, no_drive, top)
rewards[0, 1, 1] = 1.0  # (top, no_drive, rolling_down)
rewards[0, 1, 2] = 0.0  # (top, no_drive, bottom)
rewards[1, 0, 0] = 2.0  # (rolling_down, drive, top)
rewards[1, 0, 1] = 1.5  # (rolling_down, drive, rolling_down)
rewards[1, 0, 2] = 0.5  # (rolling_down, drive, bottom)
rewards[1, 1, 0] = 0.0  # (rolling_down, no_drive, top)
rewards[1, 1, 1] = 0.0  # (rolling_down, no_drive, rolling_down)
rewards[1, 1, 2] = 1.0  # (rolling_down, no_drive, bottom)
rewards[2, 0, 0] = 2.0  # (bottom, drive, top)
rewards[2, 0, 1] = 0.0  # (bottom, drive, rolling_down)
rewards[2, 0, 2] = 2.0  # (bottom, drive, bottom)
rewards[2, 1, 0] = 0.0  # (bottom, no_drive, top)
rewards[2, 1, 1] = 0.0  # (bottom, no_drive, rolling_down)
rewards[2, 1, 2] = 1.0  # (bottom, no_drive, bottom)

# --------------------------------------------------
# value iteration
# --------------------------------------------------
def get_optimal_policy(values, transitions, rewards, discount_factor):
    '''
    once the value iteration is converved, this function is used to get
    the optimal policy based on converged v(s) values
    '''
    q_values = np.zeros((num_states, num_actions))

    # compute q(s, a) for all states and actions
    for s in range(num_states):
        for action in range(num_actions):
            q_values[s, action] = sum(transitions[s, action, s_next] * (rewards[s, 
action, s_next] + discount_factor * values[s_next]) for s_next in range(num_states))

    # select the best actions as the policy for that state
    policy = np.argmax(q_values, axis=1)
    return policy
```

```python
# value iteration
def value_iteration(transitions, rewards, discount_factor, tolerance=1e-6):
    values = np.zeros(num_states)

    while(True):
        delta = 0
        for s in range(num_states):
            v = values[s]

            # update value of current state from optimal v values of next states (from last iteration)
            values[s] = max([sum(transitions[s, action, s_next] * (rewards[s, action, s_next]+discount_factor*values[s_next]) for s_next in range(num_states)) for action in range(num_actions)])

            delta = max(delta, abs(v-values[s]))

        # if values converge, break
        if delta < tolerance:
            break

    # once value iteration is converged, get the best policy
    policy = get_optimal_policy(values, transitions, rewards, discount_factor)

    return values, policy

# -------------------------------------------------
# Example usage - Value iteration with changed rewards
# -------------------------------------------------
# changing rewards (rewarding drive action more during rolling down of the car)
rewards[1, 0, 0] = 4.0  # (rolling_down, drive, top)
rewards[1, 0, 1] = 1.5  # (rolling_down, drive, rolling_down)
rewards[1, 0, 2] = 0.5  # (rolling_down, drive, bottom)

values, policy = value_iteration(transitions, rewards, discount_factor)

for i in range(num_states):
    print(f'Optimal value for state "{state_names[i]}":', np.round(values[i], 2))
    print(f'Policy for state "{state_names[i]}":', action_names[policy[i]])
    print('---------------------------------------')
```