CS 695: Programming assignment (P4)

Md. Ridwan Hossain Talukder
Department of Computer Science
George Mason University
Fairfax, USA
mtalukd@gmu.edu

I. Evaluating Policies

A. First-Visit MC Policy Evaluation

```
def evaluate_policy_first_visit_mc(world, policy, num_iterations=1000, gamma=0.98, num_steps=200, seed=695):
    """Returns a list such that value[state] is the value of that state."""
    # Perform a bunch of rollouts
    random.seed(seed)
    np.random.seed(seed)
    returns = {s: [] for s in world.states}

    for _ in range(num_iterations):
        # Rollout
        state = world.get_random_state()
        steps = []
        states_so_far = set()
        for _ in range(num_steps):
            action = policy[state]
            reward, new_state = world.execute_action(state, action)
            steps.append((state, action, reward, states_so_far.copy()))
            states_so_far.add(new_state)
            state = new_state

        G = 0
        for s, a, r, states_so_far in reversed(steps):
            G = G * gamma + r
            if s not in states_so_far:
                returns[s] += [G]

    # Return the values
    values = []
    for state, returns in returns.items():
        if len(returns) == 0:
            values.append(0)
        else:
            values.append(sum(returns)/len(returns))
    return values
```

```
== Policy 1 ==
 Chance:  0.0 | Average Value: -49.12060266971388
 Chance:  0.2 | Average Value: -48.89508342835458
 Chance:  0.5 | Average Value: -48.48344138693204
 Chance:  0.8 | Average Value: -47.71995137245433
 Chance:  1.0 | Average Value: -46.80535428375775
== Policy 2 ==
 Chance:  0.0 | Average Value: 15.232148341510536
 Chance:  0.2 | Average Value: 20.74578379235873
 Chance:  0.5 | Average Value: -12.220567532782493
 Chance:  0.8 | Average Value: -36.609129892592826
 Chance:  1.0 | Average Value: -46.80535428375775
```

1) Answer: The costs of each policy becomes the same as the random move chance becomes 1.0. When the random move chance is 1.0, that means for each state-action pair the probability is now 1.0. So the episodes generated by the both the policies become the same as well (as both policies are now acting like a random policy and fixing the random seed provides both policy with same random states). That's why the average value is also the same.

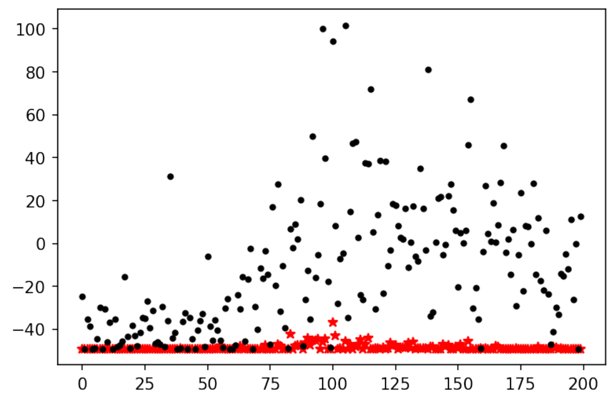

Fig. 1: Computed value for each policy (for a random move chance of 0.5)

B. Linear Algebra Solution

```
def evaluate_policy(env, policy, gamma=0.98):
    """Returns a list of values[state]."""
    states = env.states
    rewards = env.rewards

    N = len(policy)
    i = np.eye(N)
    p = np.eye(N)
    for state in states:
        p[state] = env.get_transition_probs(state, policy[state])
    a = gamma * p
    A = i - a
    x = np.linalg.solve(A,rewards)
    return x
```

```
== Policy 1 ==
 Chance:  0.0 | Average Value: -49.944999999999936
 Chance:  0.2 | Average Value: -49.67471869687554
 Chance:  0.5 | Average Value: -49.23657274697364
 Chance:  0.8 | Average Value: -48.46321813637229
 Chance:  1.0 | Average Value: -47.55915548093535
== Policy 2 ==
 Chance:  0.0 | Average Value: 14.35308340966287
 Chance:  0.2 | Average Value: 22.61595156055575
 Chance:  0.5 | Average Value: -13.931551639369333
```

```
Chance:  0.8 | Average Value: -37.611467812607856
Chance:  1.0 | Average Value: -47.55915548093535
```

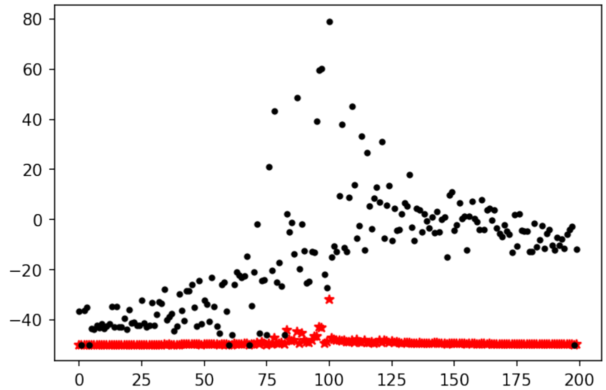

Fig. 2: Computed value for each policy (for a random move chance of 0.5)

1) Answer: The costs of each policy becomes the same as the random move chance becomes 1.0. When the random move chance is 1.0, that means for each state-action pair the probability is now 1.0. So the episodes generated by the both the policies become the same as well (as both policies are now acting like a random policy and fixing the random seed provides both policy with same random states). That's why the average value is also the same.

II. Value Iteration

```python
def value_iteration(world, num_iterations, gamma=0.98):
    # values is a vector containing the value for each state.
    values = world.rewards.copy()
    # values_over_iterations is a vector of the values at each iteration (for visualization)
    values_over_iterations = [values.copy()]
    epsilon = .5

    for _ in range(num_iterations):
        # Perform one step of value iteration
        for state in world.states:
            temp = []
            for action in world.get_actions_for_state(state):
                temp.append(np.dot(world.get_transition_probs(state, action), values))
            values[state] = world.rewards[state] + max(temp) * gamma
        if max(abs(values_over_iterations[-1]-values)) < epsilon:
            break
        # Store the values in the 'all_values' list (for visualization)
        values_over_iterations.append(values.copy())

    # Return the all_values list; all_values[-1] are the final values.
    return values_over_iterations


def compute_policy_from_values(world, values):
    """policy is a mapping from states -> actions.
    Here, it's just a vector: action = policy[state]
    """
    # Initialize the policy vector
    policy = np.zeros_like(world.states)

    # Compute the policy for every state
    for state in world.states:
        max_action = -999
        temp_val = -999
        for action in world.get_actions_for_state(state):
            if temp_val < np.dot(world.get_transition_probs(state, action), values):
                temp_val = np.dot(world.get_transition_probs(state, action), values)
                max_action = action
        policy[state] = max_action

    return policy
```

```
== Policy 1 ==
 Chance:  0.0 | Average Value: -49.944999999999936
 Chance:  0.2 | Average Value: -49.67471869687554
 Chance:  0.5 | Average Value: -49.23657274697364
 Chance:  0.8 | Average Value: -48.46321813637229
 Chance:  1.0 | Average Value: -47.55915548093535

== Policy 2 ==
 Chance:  0.0 | Average Value: 14.35308340966287
 Chance:  0.2 | Average Value: 22.61595156055575
 Chance:  0.5 | Average Value: -13.931551639369333
 Chance:  0.8 | Average Value: -37.611467812607856
 Chance:  1.0 | Average Value: -47.55915548093535

== Policy Value Iteration (rmc = 0.2) ==
 Chance:  0.0 | Average Value: 423.8196062104148
 Chance:  0.2 | Average Value: 330.7494726654674
 Chance:  0.5 | Average Value: 155.7468615368327
 Chance:  0.8 | Average Value: -8.875513345905023
 Chance:  1.0 | Average Value: -47.55915548093535
```

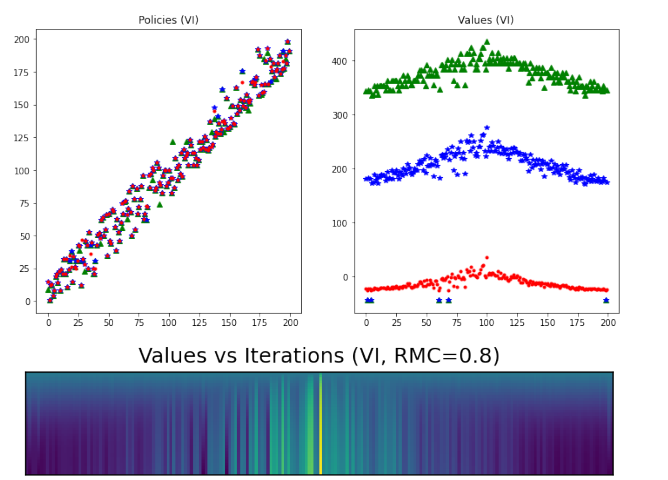

Fig. 3: Plots from value iterations

1) Answer: There are some states that have a self returning action or there are no out going actions that can lead to a better choice state, so the value does not get updated after each iterations that's why the value for those states are very low regardless of the random move chance.

III. Policy Iteration

```python
def policy_iteration(world, num_iterations, gamma=0.99):
    values = world.rewards.copy()
    all_values = []
    all_values.append(values.copy())
    # Get random policy
    policy = [random.choice(world.get_actions_for_state(state))
              for state in world.states]
    prev_policy = policy.copy()
    for ii in range(num_iterations):
        # Update the values (using solution)
        values = evaluate_policy(world, policy)
        # Update the policy
        for state in world.states:
            temp_val = [np.dot(world.get_transition_probs(state, action), values) for action in world.get_actions_for_state(state)]
            policy[state] = world.get_actions_for_state(state)[np.argmax(temp_val)]
        all_values.append(values.copy())
        # Terminate (break) if the policy does not change between steps
        if (prev_policy == policy):
            break
        prev_policy = policy.copy()
    return policy, all_values
```

```
Value Iteration Time: 0.34607648849487305
Policy Iteration Time: 0.04205822944641113
Value Iteration Avg. Value: 155.44895381214943
Policy Iteration Avg. Value: 155.74897560850138
```

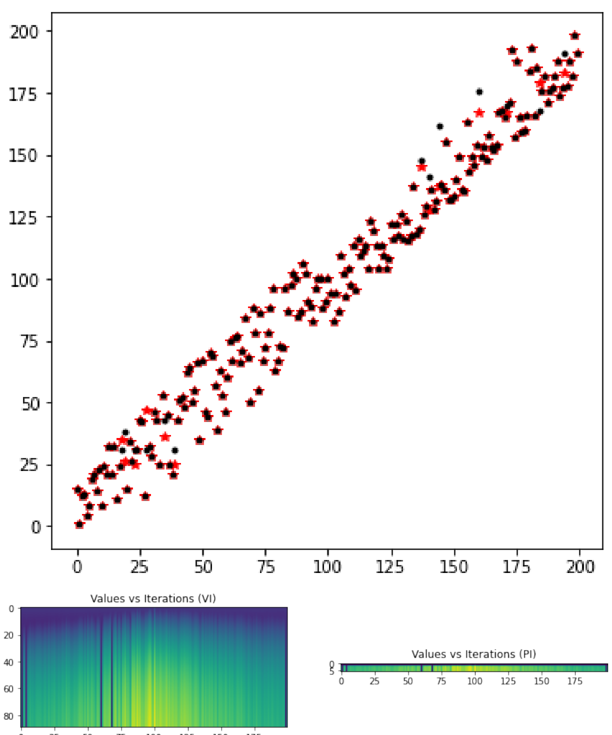

Fig. 4: Plots from policy iterations

1) Answer: Between these two, Policy Iteration converges more quickly. Because the policy iteration function goes through two phases in each iteration. First it evaluates the policy, and then it improves it. The value iteration function covers these two phases by taking a maximum over the utility function for all possible actions.

IV. Q Learning

```python
def Q_learning(env, num_iterations, num_steps=30, gamma=0.98, learning_rate=0.005, epsilon=0.1, seed=695):
    random.seed(seed)
    np.random.seed(seed)
    Q_s_a = np.zeros((len(env.states), len(env.states)))
    total_rewards = []
    # Iterate
    for ii in range(num_iterations):
        # Rollout
        total_reward = 0
        state = env.get_random_state()
        for ii in range(num_steps):
            # Take an action and get the reward
            actions = env.get_actions_for_state(state)
            if random.random() > epsilon:
                action_ind = np.argmax(Q_s_a[state, actions])
                action = actions[action_ind]
            else:
                action = random.choice(actions)
            r, new_state = env.execute_action(state, action)
            new_actions = env.get_actions_for_state(new_state)
            # Q_s_a[state, action] = (1-learning_rate) * Q_s_a[state, action] + learning_rate * (r + gamma * max(Q_s_a[new_state, new_actions]))
            Q_s_a[state, action] += learning_rate * (r + (gamma * np.max(Q_s_a[new_state, new_actions])) - Q_s_a[state, action])
            # Update reward and state
            total_reward += r
            state = new_state

        total_rewards.append(total_reward)

    policy = np.zeros(len(env.states))
    for state in env.states:
        actions = env.get_actions_for_state(state)
        action_ind = np.argmax(Q_s_a[state, actions])
        policy[state] = actions[action_ind]
    return list(policy.astype(int)), total_rewards
```

```
Policy Iteration Avg. Value: 279.70336619955066
Q Learning Avg. Value (0.001): 159.43787211551074
Q Learning Avg. Value (0.005): 245.57602303534043
Q Learning Avg. Value (0.02): 260.4733408376938
Q Learning Avg. Value (0.1): 229.03788753554144
Q Learning Avg. Value (1.0): -45.77301680451045
```

1) Answer: Q learning has to learn the transition probabilities where value iteration has the knowledge about it. Also value iteration has access to the policies, but Q learning does not. So value iteration uses the policies and transition probabilities which helps it to converge faster than Q learning.

2) Answer: In Q-learning, the experience learned by the agent is stored in the Q table, and the value in the table expresses the long-term reward value of taking specific action in a specific state. According to the Q table, the Q learning

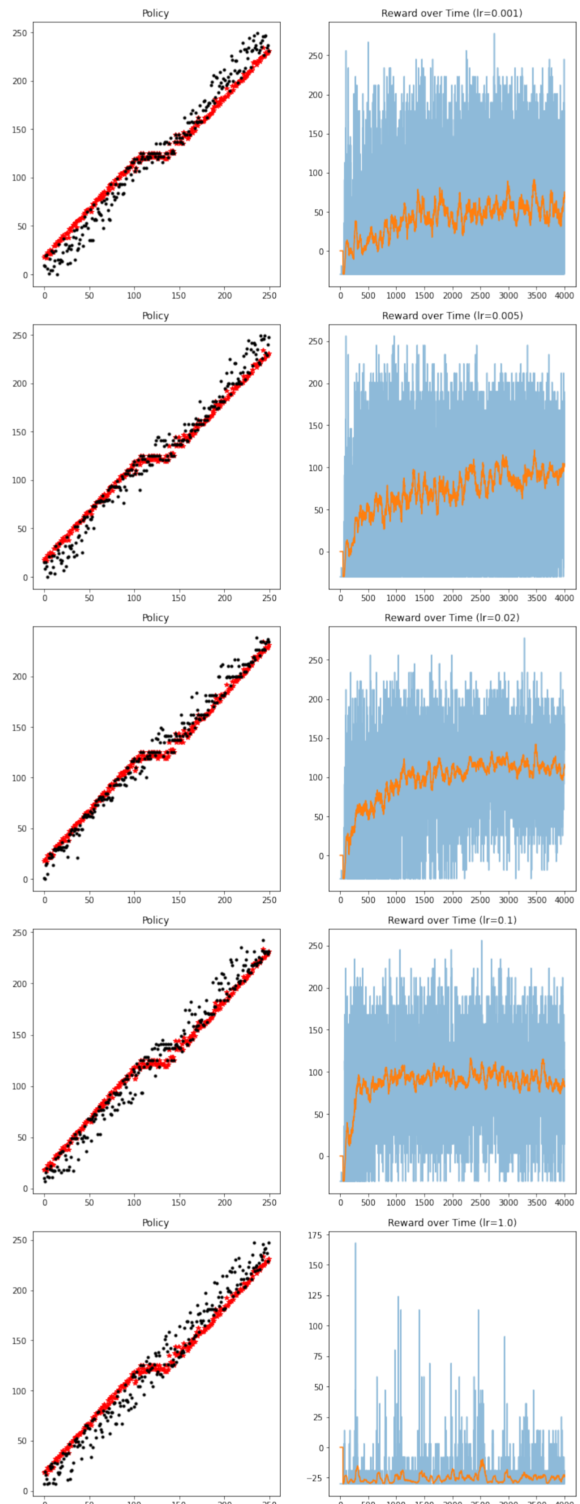

Fig. 5: Plots from Q Learning

algorithm can tell the Q agent which action to choose in a specific situation to get the largest expected reward. As in this case it does not have required learning steps, thus it's not converging to the optimal policy values. If we let the Q learning learn over more iterations it will fix the issue without changing the learning rate.

3) Answer: A learning rate that is too large can cause the model to converge too quickly to a sub-optimal solution. This is why the performance is not very good when the learning rate is too high.