CS640 Homework 5: MDP, RL and Adversarial Search

This assignment covers three topics shown in the title. There is a problem for each topic.

In particular, the MDP problem asks you to implement both the value iteration and the policy iteration, the RL problem asks you to implement the Q-learning algorithm, and in the adversarial search problem, you need to run the min-max algorithm by hand.

Collaboration

You must answer all questions **independently**, including the coding ones.

Instructions

General Instructions

In an ipython notebook, to run code in a cell or to render Markdown+LaTeX press `Ctrl+Enter` or `[>|]` (like "play") button above. To edit any code or text cell (double) click on its content. To change cell type, choose "Markdown" or "Code" in the drop-down menu above.

Most of the written questions are followed up a cell for you enter your answers. Please enter your answers in a new line below the **Answer** mark. If you do not see such cell, please insert one by yourself. Your answers and the questions should **not** be in the same cell.

Instructions on Math

Some questions require you to enter math expressions. To enter your solutions, put down your derivations into the corresponding cells below using LaTeX. Show all steps when proving statements. If you are not familiar with LaTeX, you should look at some tutorials and at the examples listed below between $..$. The OEIS website can also be helpful.

Alternatively, you can scan your work from paper and insert the image(s) in a text cell.

Submission

Once you are ready, save the note book as PDF file (File -> Print -> Save as PDF) and submit via Gradescope.

Please select pages to match the questions on Gradescope. **You may be subject to a 5% penalty if you do not do so**.

Q1: MDP

Q1.0: Setups

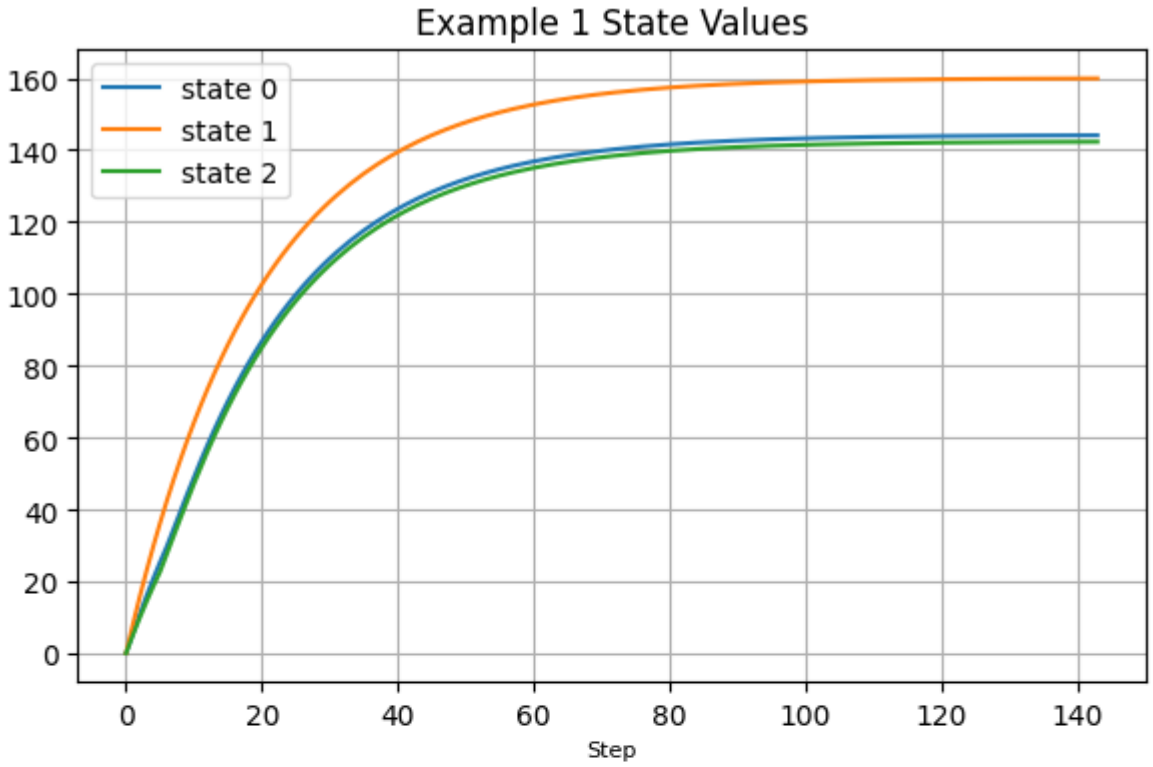
[ ] ↳ *4 cells hidden*

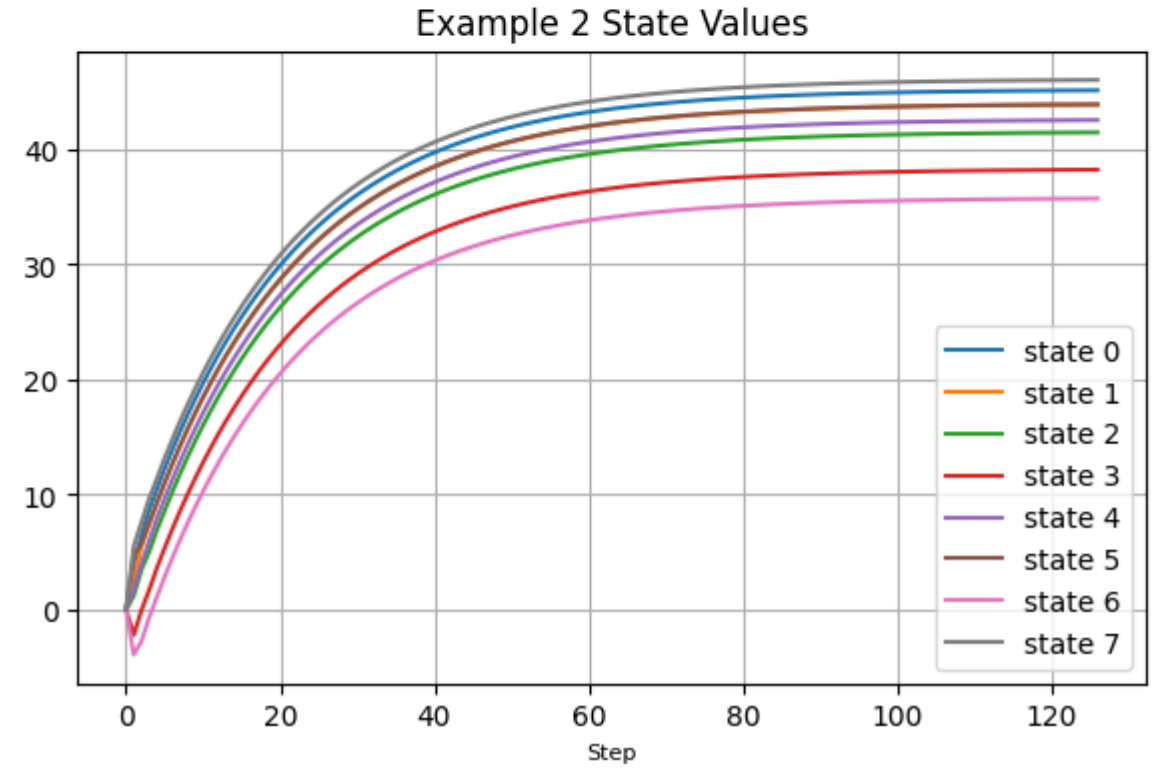
Q1.1: Value iteration

Implement value iteration by finishing the following function, and then run the cell.

```
def value_iteration(states, actions, T, R, gamma = 0.95, tolerance = 1e-2, max_steps = 10
    Vs = [] # all state values
    Vs.append(np.zeros(len(states))) # initial state values
    steps, convergent = 0, False
    while not convergent and steps < max_steps: # complete this loop
        ############################ start of your code ##########################
        # compute new state values as an array, and append the array to the list Vs
        V = -np.ones(len(states)) * sys.maxsize
        for s in states:
            for a in actions:
                new_value = np.sum(T[s, a] * (R[s, a] + gamma * Vs[-1]))
                V[s] = max(V[s], new_value)
        Vs.append(V)

        ############################ end of your code ############################
        convergent = np.linalg.norm(Vs[-1] - Vs[-2]) < tolerance
        steps += 1
    if steps == max_steps:
        print("Max iterations reached before convergence.")
        sys.exit(1)
    ############################ start of your code ##########################
    # compute Q from the last V array
    # extract the optimal policy and name it "policy" to return
    Q = np.zeros((len(states), len(actions)))
    for s in states:
        for a in actions:
            Q[s, a] = np.sum(T[s, a] * (R[s, a] + gamma * Vs[-1]))
    policy = np.argmax(Q, axis = 1)
    ############################ end of your code ############################
    return np.array(Vs), policy, steps

states, actions, T, R = example_1
gamma, tolerance, max_steps = 0.95, 1e-2, 1000
Vs_1, policy_1, steps_1 = value_iteration(states, actions, T, R, gamma, tolerance, max_st

states, actions, T, R = example_2
gamma, tolerance, max_steps = 0.95, 1e-2, 1000
Vs_2, policy_2, steps_2 = value_iteration(states, actions, T, R, gamma, tolerance, max_st

fig, axes = plt.subplots(2, 1, figsize = (6, 8))
for i in range(Vs_1.shape[1]):
    axes[0].plot(Vs_1[:, i], label = "state " + str(i))
axes[0].set_title("Example 1 State Values", fontsize = 12)
axes[0].set_xlabel("Step", fontsize = 8)
axes[0].legend()
axes[0].grid(True)
for i in range(Vs_2.shape[1]):
    axes[1].plot(Vs_2[:, i], label = "state " + str(i))
axes[1].set_title("Example 2 State Values", fontsize = 12)
axes[1].set_xlabel("Step", fontsize = 8)
axes[1].legend()
```

```
52 axes[1].grid(True)
53 fig.tight_layout()
54 plt.show()
55
56 print("Optimal policy for example 1: " + str(policy_1))
57 print("Optimal policy for example 2: " + str(policy_2))
```






```
Optimal policy for example 1: [1 0 1]
Optimal policy for example 2: [3 4 4 3 1 1 4 1]
```

Q1.2: Policy iteration

Implement policy iteration by finishing the following function, and then run the cell.

```
def policy_iteration(states, actions, T, R, gamma = 0.95, tolerance = 1e-2, max_steps = 1
    policy_list = [] # all policies explored
    initial_policy = np.array([np.random.choice(actions) for s in states]) # random polic
    policy_list.append(initial_policy)
    Vs = [] # all state values
    Vs = [np.zeros(len(states))] # initial state values
    steps, convergent = 0, False
    while not convergent and steps < max_steps:
        ########################### start of your code ########################
        # 1. Evaluate the current policy, and append the state values to the list Vs
        convergent_eval = False
        V = Vs[-1]
        while not convergent_eval:
            V_new = np.zeros(len(states))
            for s in states:
                a = policy_list[-1][s]
                Q = 0
                for s_next in states:
                    Q += T[s, a, s_next] * (R[s, a, s_next] + gamma * V[s_next])
                V_new[s] = Q
            convergent_eval = np.linalg.norm(V_new - V) < tolerance
            V = V_new
        Vs.append(V_new)

        # 2. Extract the new policy, and append the new policy to the list policy_list
        Q_values = -np.ones((len(states), len(actions))) * sys.maxsize
        for s in states:
            for a in actions:
                Q_values[s, a] = np.sum(T[s, a] * (R[s, a] + gamma * Vs[-1]))
        new_policy = np.argmax(Q_values, axis = 1)
        policy_list.append(new_policy)
        ############################ end of your code ##########################
        steps += 1
        convergent = (policy_list[-1] == policy_list[-2]).all()
    if steps == max_steps:
        print("Max iterations reached before convergence.")
        sys.exit(1)
    return Vs, policy_list, steps

print("Example MDP 1")
states, actions, T, R = example_1
gamma, tolerance, max_steps = 0.95, 1e-2, 1000
Vs, policy_list, steps = policy_iteration(states, actions, T, R, gamma, tolerance, max_st
for i in range(steps):
    print("Step " + str(i))
    print("state values: " + str(Vs[i]))
    print("policy: " + str(policy_list[i]))
    print()
print()
print("Example MDP 2")
states, actions, T, R = example_2
```

```
gamma, tolerance, max_steps = 0.95, 1e-2, 1000
Vs, policy_list, steps = policy_iteration(states, actions, T, R, gamma, tolerance, max_st
for i in range(steps):
    print("Step " + str(i))
    print("state values: " + str(Vs[i]))
    print("policy: " + str(policy_list[i]))
    print()
```

```
Example MDP 1
Step 0
state values: [0. 0. 0.]
policy: [1 1 1]

Step 1
state values: [43.65839589 40.5921489  39.70489023]
policy: [1 0 0]

Step 2
state values: [112.95002513 159.85584651  99.92720955]
policy: [1 0 1]


Example MDP 2
Step 0
state values: [0. 0. 0. 0. 0. 0. 0. 0.]
policy: [2 4 2 0 3 3 2 0]

Step 1
state values: [ -7.51257276  -3.62936276  -9.23621336  -9.75080134  -8.63433687
  -5.99069045 -14.67723454  -3.83794082]
policy: [3 4 4 3 1 1 4 1]
```

Q1.3: More tests

The following block tests both of your implementations with even more random MDPs. Simply run the cell.

[ ] ↳ *1 cell hidden*

Q2: RL

Q2.0: Setups

[ ] ↳ *13 cells hidden*

Q2.1: Q-learning

Implement Q-learning

Implement Q-learning by modifying the specified part in the following block. Currently it is only taking a random action.

```
def Q_learning(env, episodes = 100000, alpha = 0.1, gamma = 0.6, epsilon = 0.1):
    Q_values = np.zeros([env.observation_space.n, env.action_space.n])
    rewards_list = [0] * episodes
    for episode in range(episodes):
        state = env.reset()
        done = False
        while not done:
            ################################################################################
            # TO DO: implement Q-learning update
            if np.random.uniform(0, 1) < epsilon:
                action = env.action_space.sample()
            else:
                action = np.argmax(Q_values[state])
            next_state, reward, done, _, info = env.step(action) # you shouldn't change th
            Q_values[state, action] = (1 - alpha) * Q_values[state, action] + alpha * (rew
            ############################## End of your code ##########################
            rewards_list[episode] += reward
            state = next_state
    return Q_values, rewards_list
```

Q3

Q3.1

6, 6, 3, 6, 10, 3, 8, 20, 6, 10, 15, 7, 3, 16, 8

Q3.2

0-1-3-8-18

Q3.3

6, 13, 14, 20, 22, 27, 28, 29, 30