National Taiwan University
Department of Electrical Engineering
Name: Hsin-Chen, Lin
ID: B10901041

113-1 (Fall 2024)
Reinforcement Learning

Assignment 1 Report

Q1. What methods have you tried for async DP? Compare their performance.

1. Comparison of three asynchronous DP methods

In-place DP	Prioritized Sweeping	Real-time DP
1144 steps	**1288 steps**	**1567 steps**

   - **In-place dynamic programming**: This method converged the fastest, taking only 1144 steps. It avoids additional memory use by updating values directly in place, but might update the same state multiple times.
   - **Prioritized sweeping**: This method is more efficient in focusing updates on high-priority states. It balances computation and prioritization but requires maintaining a priority queue.
   - **Real-time dynamic programming (RTDP)**: This method took the longest, with 1567 steps. RTDP optimizes decisions in real-time but requires significant interaction with the environment, leading to slower convergence.

2. Method Pseudocode

   (a) In-place dynamic programming

```
Initialize values for all states arbitrarily
repeat
    Set δ ← 0
    for each state in the state space do
        v ← values[state]
        Update values[state] ← max(Q-value(state, action)) for all possible actions
        Update δ ← max(δ, |v − values[state]|)
    end for
until δ < threshold
for each state in the state space do
    policy[state] ← arg max(Q-value(state, action))
end for
```

   (b) Prioritized sweeping

```
Initialize values for all states arbitrarily
Create an empty priority queue
for each state in the state space do
    Compute the error: error ← |get_state_value(state) − values[state]|
    if error > θ then
        Push state into priority queue with negative priority
    end if
end for
while priority queue is not empty do
    if policy_evaluation() < threshold then
        exit loop
    end if
    Pop the state with the highest priority from the queue
    Update values[state] ← get_state_value(state)
    for each predecessor of the state do
        Store the old value of the predecessor
```

```
            Update values[predecessor] ← get_state_value(predecessor)
            Compute the new error for the predecessor: pre_error ← |new_value − old_value|
            if pre_error > θ then
                Push predecessor into the priority queue
            end if
        end for
    end while
    Perform policy improvement
```

(c) Real-time dynamic programming

```
for each state in the state space do
    if policy_evaluation() < threshold then
        exit loop
    end if
    while True do
        Select action: action ← arg max(Q-value(state, action))
        Update values[state] ← get_state_value(state)
        Take a step: (next_state, reward, done) ← grid_world.step(state, action)
        if done then
            break
        end if
        for each predecessor of the state do
            Update values[predecessor] ← get_state_value(predecessor)
        end for
        state ← next_state
    end while
end for
policy_improvement()
```

Q2. What is your final method? How is it better than other methods you've tried?

1. My final method is In-place Dynamic Programming. This method is simple since it avoids the need for extra data structures like priority queues and that for tracking predecessors as well as future states continuously during interaction with the environment. While this method directly updates the value of each state based on the immediate best action for that state, it quickly reduces the difference between continuous value estimates. With all of the reasons mentioned, In-place Dynamic Programming is better than other methods I've tried.

2. Tried a novel method aside from the 3 method mentioned in class, but the step number 2193 is much larger than the others.

```
Sort the states based on values in descending order
for each sorted state do
    if policy_evaluation() < threshold then
        exit loop
    end if
    while True do
        Select action: action ← arg max(Q-value(state, action))
        Update values[state] ← get_state_value(state)
        Take a step: (next_state, reward, done) ← grid_world.step(state, action)
        if done then
            break
        end if
        for each predecessor of the state do
            Update values[predecessor] ← get_state_value(predecessor)
        end for
        state ← next_state
```

```
    end while
end for
policy_improvement()
```