Reinforcement Learning Assignment 1

Chunlin Chen, 519021910463

March 14, 2023

1 Problem Description

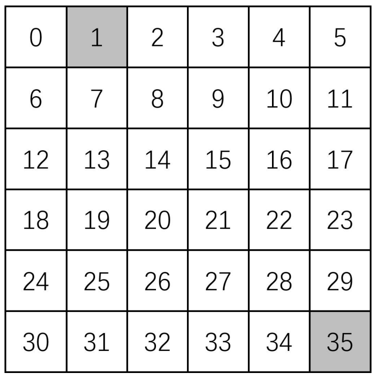


Figure 1: Gridworld

Solve the Gridworld problem illustrated in Figure 1 using Dynamic Programming Algorithms, including:

- Iterative Policy Evaluation of a uniform random policy: $\pi(n|\cdot) = \pi(e|\cdot) = \pi(s|\cdot) = \pi(w|\cdot) = 0.25$.
- Improving the above policy using Policy Iteration.
- Improving the above policy using Value Iteration.

Let the discount factor γ be 1.

2 Implementation of DP Algorithms

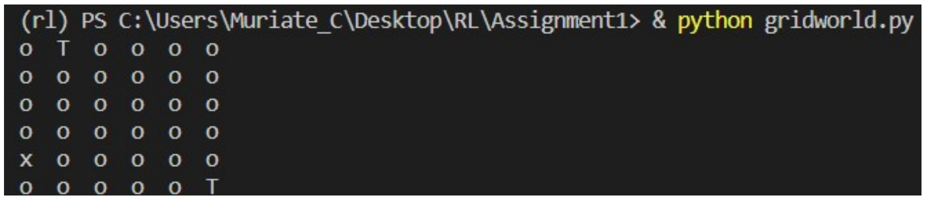
2.1 Build the Gridworld Environment

We create the Gridworld by using a class. The class 'GridWorld' is defined as below:

```
class GridWorld(DiscreteEnv):
   metadata = {'render.modes': ['human', 'ansi']}

   def __init__(self, shape=[6,6]):
      if not isinstance(shape, (list, tuple)) or not len(shape) == 2:
         raise ValueError('shape argument must be a list/tuple of
             length 2')

      self.shape = shape

      nS = np.prod(shape)
      nA = 4

      N = 0
      E = 1
      S = 2
      W = 3

      MAX_Y = shape[0]
      MAX_X = shape[1]

      P = {}
      grid = np.arange(nS).reshape(shape)
      it = np.nditer(grid, flags=['multi_index'])

      while not it.finished:
         s = it.iterindex
         y, x = it.multi_index

         # P[s][a] = (prob, next_state, reward, is_done)
         P[s] = {a : [] for a in range(nA)}

         is_done = lambda s: s == 1 or s == (nS - 1)
         reward = 0.0 if is_done(s) else -1.0

         # terminal state
         if is_done(s):
            P[s][N] = [(1.0, s, reward, True)]
            P[s][E] = [(1.0, s, reward, True)]
            P[s][S] = [(1.0, s, reward, True)]
            P[s][W] = [(1.0, s, reward, True)]
         # non-terminal state
         else:
            next_s_north = s if y == 0 else s - MAX_X
            next_s_east = s if x == (MAX_X - 1) else s + 1
            next_s_south = s if y == (MAX_Y - 1) else s + MAX_X
            next_s_west = s if x == 0 else s - 1
            P[s][N] = [(1.0, next_s_north, reward,
                is_done(next_s_north))]
            P[s][E] = [(1.0, next_s_east, reward,
                is_done(next_s_east))]
            P[s][S] = [(1.0, next_s_south, reward,
                is_done(next_s_south))]
            P[s][W] = [(1.0, next_s_west, reward,
                is_done(next_s_west))]

        it.iternext()

    # Initial state distribution is uniform
    isd = np.ones(nS) / nS

    self.P = P

    super(GridWorld, self).__init__(nS, nA, P, isd)
```

nS and nA denote the number of states and actions. N, E, S and W denote the actions North, East, South and West. P is a dictionary which stores the state transition probability matrix, and grid is a numpy ndarray which stores each state of the grid. In the while loop, we write the state transition rules for the 4 actions taken in each state and store the information into the dictionary P. In this way, we have defined the necessary elements of the class 'GridWorld'.

Here we render the $6x6$ grid in the terminal:



```
(rl) PS C:\Users\Muriate_C\Desktop\RL\Assignment1> & python gridworld.py
o  T  o  o  o  o
o  o  o  o  o  o
o  o  o  o  o  o
o  o  o  o  o  o
x  o  o  o  o  o
o  o  o  o  o  T
```

Figure 2: The 6x6 Gridworld Environment

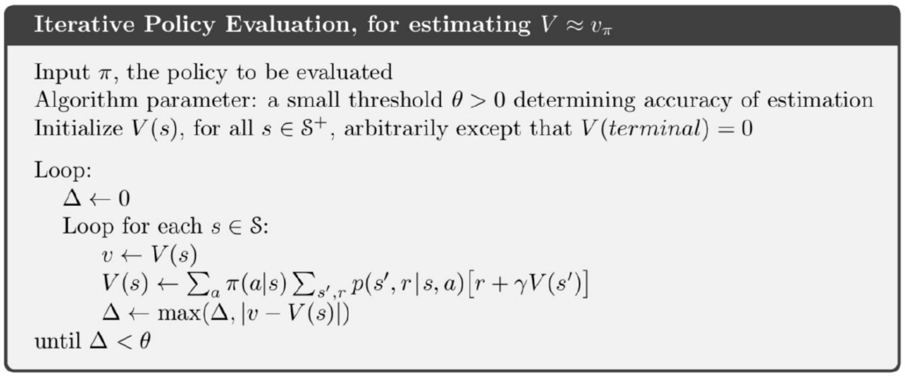
As shown in Figure 2, 'T' represents the terminal state, 'X' represents the current position of the agent, and 'O' represents the regular state.

2.2 Iterative Policy Evaluation

According to the pseudocode of Iterative Policy Evaluation shown in Figure 3, we can implement the algorithm in a function like this:

```
def policy_eval(policy, env, discount_factor=1.0, theta=1e-5):
    V = np.zeros(env.nS)
    while True:
        delta = 0
        for s in range(env.nS):
            v = 0
            for a, action_prob in enumerate(policy[s]):
                for prob, next_state, reward, done in env.P[s][a]:
                    v += action_prob * prob * (reward + discount_factor *
                        V[next_state])
            delta = max(delta, np.abs(v - V[s]))
            V[s] = v

        if delta < theta:
            break
    return np.array(V)
```

Here we let the θ be 1^{-5}.

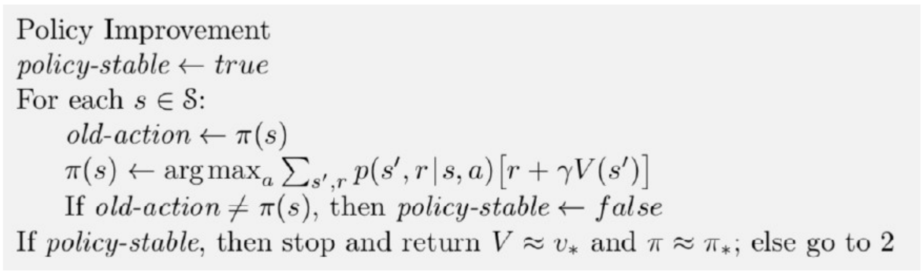


```
Iterative Policy Evaluation, for estimating $V \approx v_\pi$

Input $\pi$, the policy to be evaluated
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
   $\Delta \leftarrow 0$
   Loop for each $s \in \mathcal{S}$:
      $v \leftarrow V(s)$
      $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
      $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$
```

Figure 3: The Pseudocode of Iterative Policy Evaluation

2.3 Policy Iteration

On the basis of Iteration Policy Evaluation, we can implement Policy Improvement shown in Figure 4 and realize the Policy Iteration Algorithm:



```
Policy Improvement
$policy\text{-}stable \leftarrow true$
For each $s \in \mathcal{S}$:
    $old\text{-}action \leftarrow \pi(s)$
    $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
    If $old\text{-}action \neq \pi(s)$, then $policy\text{-}stable \leftarrow false$
If $policy\text{-}stable$, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2
```

Figure 4: The Pseudocode of Policy Improvement

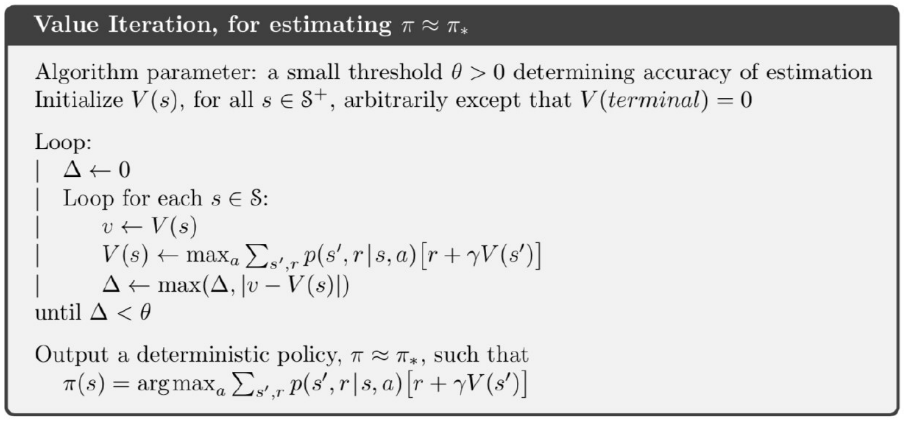
```
def policy_improvement(env, policy_eval_fn=policy_eval,
    discount_factor=1.0):
```

```
    policy = np.ones([env.nS, env.nA]) / env.nA

    while True:
        V = policy_eval_fn(policy, env, discount_factor)

        policy_stable = True

        for s in range(env.nS):
            chosen_a = np.argmax(policy[s])

            action_values = np.zeros(env.nA)
            for a in range(env.nA):
                for prob, next_state, reward, done in env.P[s][a]:
                    action_values[a] += prob * (reward + discount_factor *
                         V[next_state])
            best_a = np.argmax(action_values)

            if chosen_a != best_a:
                policy_stable = False
            policy[s] = np.eye(env.nA)[best_a]

        if policy_stable:
            return policy, V
```

2.4 Value Iteration

According to the pseudocode of Value Iteration shown in Figure 5, we can implement the algorithm in a function like this:

```
def value_iteration(env, theta=0.0001, discount_factor=1.0):
    def one_step_lookahead(state, V):
        A = np.zeros(env.nA)
        for a in range(env.nA):
            for prob, next_state, reward, done in env.P[state][a]:
                A[a] += prob * (reward + discount_factor * V[next_state])
        return A

    V = np.zeros(env.nS)
    while True:
        delta = 0

        for s in range(env.nS):
            A = one_step_lookahead(s, V)
            best_action_value = np.max(A)
            delta = max(delta, np.abs(best_action_value - V[s]))
            V[s] = best_action_value

        if delta < theta:
            break

    policy = np.zeros([env.nS, env.nA])
    for s in range(env.nS):
        A = one_step_lookahead(s, V)
        best_action = np.argmax(A)
        policy[s, best_action] = 1.0

    return policy, V
```



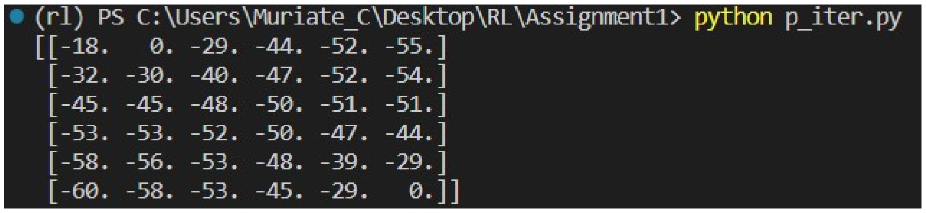
```
Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
|   $\Delta \leftarrow 0$
|   Loop for each $s \in \mathcal{S}$:
|       $v \leftarrow V(s)$
|       $V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
|       $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
    $\pi(s) = \arg\max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
```

Figure 5: The Pseudocode of Value Iteration

3 Results Analysis

3.1 Evaluation of the Uniform Random Policy

We run the Iterative Policy Evaluation on the uniform random policy, and the value function under this policy is shown in Figure 6:



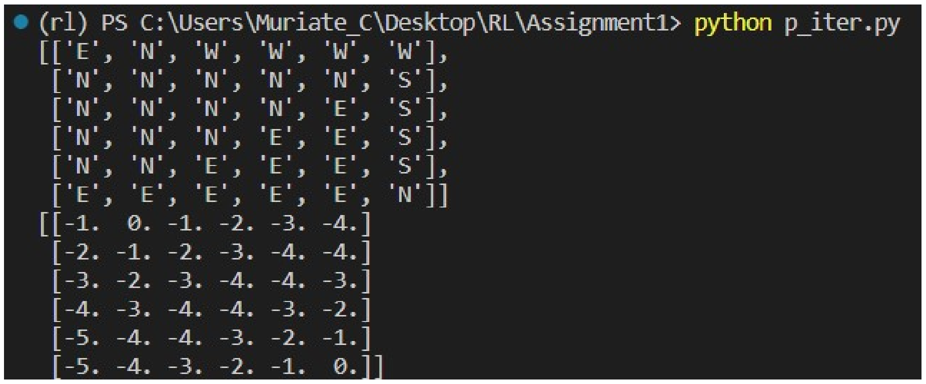
```
(rl) PS C:\Users\Muriate_C\Desktop\RL\Assignment1> python p_iter.py
[[-18.   0. -29. -44. -52. -55.]
 [-32. -30. -40. -47. -52. -54.]
 [-45. -45. -48. -50. -51. -51.]
 [-53. -53. -52. -50. -47. -44.]
 [-58. -56. -53. -48. -39. -29.]
 [-60. -58. -53. -45. -29.   0.]]
```

Figure 6: Evaluation of the Uniform Random Policy

Obviously the closer to the terminal states, the bigger the values of the states.

3.2 Policy Improvement using Policy Iteration

We run the Policy Iteration to improve the initial policy, and eventually the optimal policy and value function are shown in Figure 7:

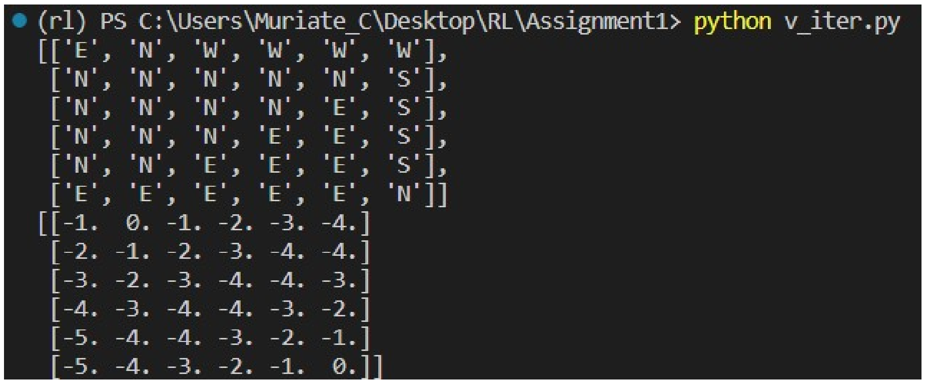


```
● (rl) PS C:\Users\Muriate_C\Desktop\RL\Assignment1> python p_iter.py
  [['E', 'N', 'W', 'W', 'W', 'W'],
   ['N', 'N', 'N', 'N', 'N', 'S'],
   ['N', 'N', 'N', 'N', 'E', 'S'],
   ['N', 'N', 'N', 'E', 'E', 'S'],
   ['N', 'N', 'E', 'E', 'E', 'S'],
   ['E', 'E', 'E', 'E', 'E', 'N']]
  [[-1.  0. -1. -2. -3. -4.]
   [-2. -1. -2. -3. -4. -4.]
   [-3. -2. -3. -4. -4. -3.]
   [-4. -3. -4. -4. -3. -2.]
   [-5. -4. -4. -3. -2. -1.]
   [-5. -4. -3. -2. -1.  0.]]
```

Figure 7: Result of Policy Iteration

3.3 Policy Improvement using Value Iteration

We run the Value Iteration to improve the initial policy, and eventually the optimal policy and value function are shown in Figure 8:



```
● (rl) PS C:\Users\Muriate_C\Desktop\RL\Assignment1> python v_iter.py
  [['E', 'N', 'W', 'W', 'W', 'W'],
   ['N', 'N', 'N', 'N', 'N', 'S'],
   ['N', 'N', 'N', 'N', 'E', 'S'],
   ['N', 'N', 'N', 'E', 'E', 'S'],
   ['N', 'N', 'E', 'E', 'E', 'S'],
   ['E', 'E', 'E', 'E', 'E', 'N']]
  [[-1.  0. -1. -2. -3. -4.]
   [-2. -1. -2. -3. -4. -4.]
   [-3. -2. -3. -4. -4. -3.]
   [-4. -3. -4. -4. -3. -2.]
   [-5. -4. -4. -3. -2. -1.]
   [-5. -4. -3. -2. -1.  0.]]
```

Figure 8: Result of Value Iteration

3.4 Comparison between the two Algorithms

From the above results, we can know that the optimal policies and value functions obtained from the two algorithms are the same. Theoretically the value iteration algorithm converges faster than the policy iteration algorithm, which, however, may not show obvious differences when solving a problem of a relatively small scale described in this report.