```
### HW1 Code

import numpy as np
import time
import cplex

"""For policy_evaluation, policy_improvement, policy_iteration and value_iteration,
the parameters P, nS, nA, gamma are defined as follows:

        P: nested dictionary
                For each pair of states in [1, nS] and actions in [1, nA], P[state][
action] is a
                tuple of the form (probability, nextstate, reward, terminal) where
                        - probability: float
                                the probability of transitioning from "state" to "ne
xtstate" with "action"
                        - nextstate: int
                                denotes the state we transition to (in range [0, nS
- 1])
                        - reward: int
                                the reward for transitioning from "state" to
                                "nextstate" with "action"
        nS: int
                number of states in the environment
        nA: int
                number of actions in the environment
        gamma: float
                Discount factor. Number in range [0, 1)
"""

class MDP:
    # Define MDP as class
    def __init__(self):
        self.nA = 2
        self.nS = 2

        self.P = {s : {a : [] for a in range(self.nA)} for s in range(self.nS)}

        # P[state][action] = (probability, nextstate, reward)
        for s in range(self.nS):
            for a in range(self.nA):
                reward = ( -2*(s==0 and a==0) -0.5*(s==0 and a==1)
                        -1*(s==1 and a==0) -3*(s==1 and a==1))

                for nextstate in range(self.nS):
                    li = self.P[s][a]
                    if nextstate == a:
                        li.append((0.75, nextstate, reward))
                    else:
                        li.append((0.25, nextstate, reward))


def policy_evaluation(P, nS, nA, policy, gamma):
    value_function = np.zeros(nS)
    value_function_new = np.zeros(nS)

    # Make R, P matrix
    R_policy = np.zeros([nS, 1])
    P_policy = np.zeros([nS, nS])

    # Initialzie diff to begin loop
    v_diff = 1

    while np.max(v_diff) > 0:
        # Perform Bellman update
        for s in range(nS):
            # Select action based on policy
            a = policy[s]

            # Initialization
            v_next = 0
            r_sum = 0
```

```
            for tup in P[s][a]:
                p, s_next, r = tup
                r_sum += p*r
                v_next += p*value_function[s_next]

            value_function_new[s] = r_sum + gamma*v_next

        # Check convergence
        v_diff = np.abs(value_function_new - value_function)

        value_function = np.copy(value_function_new)

    return value_function


def policy_improvement(P, nS, nA, value_from_policy, policy, gamma):
    # Initialization
    new_policy = np.zeros(nS, dtype='int')

    for s in range(nS):
        # Initialziation
        v_max = -1000

        for a in range(nA):
            # Intialization
            v_next = 0
            r_sum = 0

            for tup in P[s][a]:
                p, s_next, r = tup

                r_sum += p*r
                v_next += p*value_from_policy[s_next]

            v = r_sum + gamma*v_next

            # Find argmax(v)
            if v >= v_max:
                argmax_a = a
                v_max = v

        # Construct new policy
        new_policy[s] = argmax_a

    return new_policy


def policy_iteration(P, nS, nA, gamma):
    # Initialization
    value_function = np.zeros(nS)
    policy = np.zeros(nS, dtype=int)

    # intialize diff
    diff = 1

    # loop counter
    idx = 1

    while diff != 0:
        # Evaluate policy
        value_from_policy = policy_evaluation(P, nS, nA, policy, gamma)

        # Update policy
        new_policy = policy_improvement(P, nS, nA, value_from_policy, policy, gamma)

        # Calculate difference
        policy_diff = policy - new_policy
        diff = np.max(np.abs(policy_diff))

        # Update policy
        policy = np.copy(new_policy)

        # Count total iteration
```

```
        idx += 1

    value_function = policy_evaluation(P, nS, nA, policy, gamma)
    print("Total Iteration : {}".format(idx))

    return value_function, policy


def value_iteration(P, nS, nA, gamma):
    # Initialization
    value_function = np.zeros(nS)
    policy = np.zeros(nS, dtype=int)

    # Make new array to store new value function
    value_function_new = np.zeros(nS)

    # Initialize v_diff to begin vi
    v_diff = 1

    # Loop counter
    idx = 1

    while np.max(v_diff) > 0:
        for s in range(nS):
            # Intialize v_max
            v_max = -1000

            for a in range(nA):
                # Intitialization
                v_next = 0
                r_sum = 0

                for tup in P[s][a]:
                    p, s_next, r  = tup
                    r_sum += p*r
                    v_next += p*value_function[s_next]

                # Bellman equation
                v = r_sum + gamma*v_next

                if v >= v_max:
                    v_max = v
                    argmax_a = a

            # v_max is calculated. if v*, a_max will be optimal policy
            value_function_new[s] = v_max
            policy[s] = argmax_a

        # Check convergence
        v_diff = np.abs(value_function_new - value_function)

        value_function = np.copy(value_function_new)

        # Loop counter
        idx += 1

    print("Total Iteration : {}".format(idx))

    return value_function, policy


def linear_programming(P, nS, nA, gamma):

    # Form problem instance
    prob = cplex.Cplex()
    prob.objective.set_sense(prob.objective.sense.maximize)

    names = ["s1","s2"]
    obj = [-0.5, -0.5]
    lower_bounds = [-1000, -1000]
    upper_bounds = [1000, 1000]

    prob.variables.add(obj = obj,
```

```
            lb = lower_bounds,
            ub = upper_bounds,
            names = names)

    # Constraint setup
    constraint_names = ["c1_1","c1_2","c2_1","c2_2"]
    constraint_senses = ["G", "G", "G", "G"]
    constraints = []
    rhs = []

    # Constraints based on v-(gamma)Pv >= r
    for s in range(nS):
        for a in range(nA):
            r_sum = 0
            v_next = 0
            coeff = [0, 0]

            for tup in P[s][a]:
                p, s_next, r = tup
                r_sum += p*r
                coeff[s_next] = (s == s_next) -gamma*p

            constraints.append([[0,1], coeff])
            rhs.append(r_sum)

    # Input the constraints
    prob.linear_constraints.add(lin_expr = constraints,
            senses = constraint_senses,
            rhs = rhs,
            names = constraint_names)

    # Solve the problem
    prob.solve()
    V_lp = prob.solution.get_values()
    policy = np.zeros(nS, dtype=int)

    # From v*, get optimal policy
    for s in range(nS):
        q_max = -1000
        for a in range(nA):
            # Initialization
            v_next = 0
            r_sum = 0

            for tup in P[s][a]:
                p, s_next, r = tup
                r_sum += p*r
                v_next += p*V_lp[s_next]

            q = r_sum + gamma*v_next

            if q >= q_max:
                argmax_a = a
                q_max = q

        policy[s] = argmax_a


    return V_lp, policy


def relative_policy_evaluation(P, nS, nA, policy):
    # Initialization
    h = np.zeros(nS)
    #Th = np.zeros(nS)

    # Solve g + h = r + Ph, where h[0] = 0
    # g = (P*)r
    # g[1] + h[1] = r[1] + p(1|1)h[1]

    # Calculate P*
    # P*=eq
    # q is solution of q = qP, where sum of elements in q = 1
```

```
    # q(I-P) = 0
    # q[0] + q[1] = 1

    q_coeff = {s : np.zeros(nS) for s in range(nS)}
    reward = np.zeros(nS)

    for s in range(nS):
        # Select policy
        a = policy[s]

        for tup in P[s][a]:
            p, s_next, r = tup
            q_coeff[s_next][s] = (s == s_next) - p
            reward[s] += p*r

    A = np.array([q_coeff[0], [1,1]])
    B = np.array([0, 1])

    P_star = np.linalg.solve(A,B)

    g = np.dot(P_star, reward)

    # solve g[1] + h[1] = r[1] + p(1|1)h[1]
    h[1] = (reward[1] - g) / q_coeff[1][1]

    return h, g


def relative_policy_iteration(P, nS, nA):
    # Initialization
    policy = np.zeros(nS, dtype=int)

    # intialize diff
    diff = 1

    # loop counter
    idx = 1

    while diff != 0:
        # Evaluate policy
        h_from_policy, g = relative_policy_evaluation(P, nS, nA, policy)

        # Update policy : same as policy improvement step in pi
        new_policy = policy_improvement(P, nS, nA, h_from_policy, policy, gamma=1)

        # Calculate difference
        policy_diff = policy - new_policy
        diff = np.max(np.abs(policy_diff))

        # Update policy
        policy = np.copy(new_policy)

        # Count total iteration
        idx += 1

    h, g = relative_policy_evaluation(P, nS, nA, policy)

    print("Total Iteration : {}".format(idx))

    return h, policy, g


def relative_value_iteration(P, nS, nA):
    # Initialization
    h = np.zeros(nS)
    Th = np.zeros(nS)
    policy = np.zeros(nS, dtype=int)

    # intialize diff
    diff = np.ones(nS)

    # loop counter
    idx = 1
```

```
    while np.max(diff) > 0:
        # Iteration loop
        for s in range(nS):
            # Initialization
            q_max = -1000
            for a in range(nA):
                # Initialization
                h_next = 0
                r_sum = 0

                for tup in P[s][a]:
                    p, s_next, r = tup
                    r_sum += p*r
                    h_next += p*h[s_next]

                q = r_sum + h_next

                if q >= q_max:
                    q_max = q
                    argmax_a = a

            Th[s] = q_max
            policy[s] = argmax_a

        # Check convergence
        diff = np.abs(Th - Th[0] - h)

        # Update h
        h = Th - Th[0]

        # Loop counter
        idx += 1

    print("Total Iteration : {}".format(idx))

    return h, policy, Th[0]


if __name__ == "__main__":

    env = MDP()

    print("\n" + "-"*25 + "\nBeginning Policy Iteration\n" + "-"*25)

    V_pi, p_pi = policy_iteration(env.P, env.nS, env.nA, gamma=0.9)

    print("pi* : {}".format(p_pi+1))
    print("v* : {}".format(V_pi))

    print("\n" + "-"*25 + "\nBeginning Value Iteration\n" + "-"*25)

    V_vi, p_vi = value_iteration(env.P, env.nS, env.nA, gamma=0.9)

    print("pi* : {}".format(p_vi+1))
    print("v* : {}".format(V_vi))

    print("\n" + "-"*29 + "\nBeginning Linear Programming\n" + "-"*29)

    V_lp, p_lp = linear_programming(env.P, env.nS, env.nA, gamma=0.9)

    print("pi* : {}".format(p_lp+1))
    print("v* : {}".format(V_lp))

    print("\n" + "-"*35 + "\nBeginning Relative Policy Iteration\n" + "-"*35)

    h_rvi, p_rvi, g_rvi = relative_policy_iteration(env.P, env.nS, env.nA)

    print("pi* : {}".format(p_rvi+1))
    print("h* : {}".format(h_rvi))
    print("g* : {}".format(g_rvi))
```

```
    print("\n" + "-"*35 + "\nBeginning Relative Value Iteration\n" + "-"*35)

    h_rvi, p_rvi, g_rvi = relative_value_iteration(env.P, env.nS, env.nA)

    print("pi* : {}".format(p_rvi+1))
    print("h* : {}".format(h_rvi))
    print("g* : {}".format(g_rvi))
```