Problem 1: Inventory Control

The goal of this problem is to model an inventory control problem with a finite horizon Markov Decision Process (MDP) and derive the optimal solution.

Problem Description

Each month, the manager of a warehouse determines current inventory of a single product. Based on this information, he decides whether or not to order additional stock from a supplier. In doing so, he is faced with a tradeoff between the costs associated with ordering and keeping inventory and the lost sales associated with being unable to satisfy a customer demand for the product. The manager's goal is to maximize the profit. Demand for the product is random with a known probability distribution.

To see if MDP is a good model for this problem discription, you need to check if the decisions at the current time affects the future. In this case, it does because the current order will determine how many units is available to sell now and also affects how many units will remain for the future.

Problem Formulation

To model the problem as a finite horizon MDP, we need to specify a tuple (S, A, P, r, H). Suppose the decision epoch is H months. Let s_t denote the number of units in the warehouse in the beginning of month t. Suppose the capacity of the warehouse is M. Thus, the state space is $S = \{0, 1, 2, \cdots, M\}$. In state s, the manager can order at most $M - s$ units. Thus, the set of admissible controls is $A_s = \{0, 1, \cdots, M - s\}$. Suppose the demand D_t at month t has a known time-homogeneous probability distribution $p_j = \mathbb{P}(D_t = j), j = 0, 1, 2, \cdots$ and the manager orders a_t items. The system dynamics can be represented as

$$s_{t+1} = \max\{s_t + a_t - D_t, 0\}.$$

The transition probability is then given by

$$p(j \mid s, a) = \begin{cases} 0 & j > s + a \\ p_{s+a-j} & 1 \leq j \leq s + a \\ q_{s+a} & j = 0 \end{cases}$$

where $q_u = \sum_{j=u}^{\infty} p_j$ (Why?).

It remains to formulate the reward function. We need to account for three different costs/rewards: ordering costs, storage costs, and selling reward. Suppose the ordering cost $O(u)$ consists of a fixed cost and a cost growing with the order size u, i.e.,

$$O(u) = \begin{cases} K + c(u) & if u > 0 \\ 0 & if u = 0 \end{cases}$$

Let $h(u)$ be a nondecreasing function denoting the storage cost for u units and suppose there is no cost/reward at the last decision epoch. Finally, if the demand is j units and sufficient units are in the warehouse, a selling reward of $f(j)$ units is obtained. Thus, the reward function can be written as

$$r_t(s_t, a_t, s_{t+1}) = f(s_t + a_t - s_{t+1}) - h(s_t + a_t) - O(a_t).$$

It is more convenient to work with $r_t(s_t, a_t)$. To this end we compute $F_t(u)$, the expected value of revenue received in month t if there are u units in the warehouse as

$$F_t(u) = \sum_{j=0}^{u-1} f(j) p_j + f(u) q_u$$

(Why?). Thus, the reward function is

$$r_t(s, a) = F(s + a) - h(s + a) - O(a), \qquad t = 1, 2, \cdots, H - 1$$

and $r_H = 0$.

Your Job

Let $K = 4, c(u) = 2u, h(u) = u, M = 3, H = 3, f(u) = 8u$ and

$$p_j = \begin{cases} 0.25 & j = 0 \\ 0.5 & j = 1 \\ 0.25 & j = 2 \end{cases}$$

For your convenience we have hard coded the transition probability and the reward function. Your job is to implement the `optimal_policy_and_value` function using the dynamic programming algorithm to compute the optimal policy and value function. Please make sure that your code is in the designated area.

```
from __future__ import division
from __future__ import print_function
import numpy as np
```

```
class InventoryControlMDP():
    def __init__(self, H=4):
        self.H = H
        self.S = 4 # M=3 which means there are four states: 0, 1, 2, 3.
        self.terminal_reward = np.zeros(self.S)
        self.r = self._get_reward()
        self.P = self._get_transition()

    def _get_reward(self):
        r = [[0,-1,-2,-5],[5,0,-3,-np.inf],[6,-1,-np.inf,-np.inf],[5,-np.inf,-np.inf,-np.inf]]

        return np.array(r)

    def _get_transition(self):

        P = np.zeros((self.S,self.S,self.S)) ####p(s,a,s')
        p_mat = np.array([[1,0,0,0],[3/4,1/4,0,0],[1/4,1/2,1/4,0],[0,1/4,1/2,1/4]])

        for i in range(self.S):
            for j in range(self.S-i):
                for k in range(self.S):
                    P[i][j][k] = p_mat[i+j][k]
        return P

    def optimal_policy_and_value(self):
        """
        This function should return two numpy arrays denoting
        the optimal policy and value function.
        """
        policy = np.zeros((self.S, self.H-1)) # element (s, h) denotes the optimal policy at state s and time h
        value = np.zeros((self.S, self.H)) # element (s, h) denotes the value function at state s and time h
        ################ Your Code Here ###############









        ################ End of Your Code #############
        return policy, value

mdp = InventoryControlMDP(H=4)
policy, value = mdp.optimal_policy_and_value()
print("The optimal policy is")
print(policy)
print('-'*20)
print("The optimal value is")
print(value)
```

Problem 2: Frozen Lake

The goal of this problem is to get familiar with OpenAI Gym, implement value iteration and policy iteration.

Problem Description

OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking to playing games like Pong or Pinball. For more information visit https://gym.openai.com.

In this computer assigment, you'll get familiar with Frozen Lake environment and implement value and policy iteration algorithms. Frozen Lake is an environment where the agent controls the movement of a character in a grid world. Some tiles of the grid are walkable, and others lead to the agent falling into the water. Additionally, the movement direction of the agent is uncertain and only partially depends on the chosen direction. The agent is rewarded for finding a walkable path to a goal tile. For more information please visit https://gym.openai.com/envs/FrozenLake8x8-v0/.

Your Job

1. Get started with gym by following the steps here https://gym.openai.com/docs/.
2. Read https://gym.openai.com/envs/FrozenLake8x8-v0/ to get familiar with the environment, states, reward function, etc.

3. Implement the `value_iteration` function below.
4. Implement the `policy_iteration` function below.
5. Answer the questions (By double click on the cell you can edit the cell and put your answer below each question).

```
from __future__ import print_function
from __future__ import division

import numpy as np
import gym
from gym import wrappers
import time

def run_episode(env, policy, gamma, render = False):
    """ Evaluates policy by using it to run an episode and finding its
    total reward.
    args:
    env: gym environment.
    policy: the policy to be used.
    gamma: discount factor.
    render: boolean to turn rendering on/off.
    returns:
    total reward: real value of the total reward recieved by agent under policy.
    """
    obs = env.reset()
    total_reward = 0
    step_idx = 0
    while True:
        if render:
            env.render()
        obs, reward, done , _ = env.step(int(policy[obs]))
        total_reward += (gamma ** step_idx * reward)
        step_idx += 1
        if done:
            break
    return total_reward


def evaluate_policy(env, policy, gamma,  n = 100):
    """ Evaluates a policy by running it n times.
    returns:
    average total reward
    """
    scores = [
            run_episode(env, policy, gamma=gamma, render = False)
            for _ in range(n)]
    return np.mean(scores)

def extract_policy(v, gamma):
    """ Extract the policy given a value-function """
    policy = np.zeros(env.observation_space.n)
    for s in range(env.observation_space.n):
        q_sa = np.zeros(env.action_space.n)
        for a in range(env.action_space.n):
            for next_sr in env.P[s][a]:
                # next_sr is a tuple of (probability, next state, reward, done)
                p, s_, r, _ = next_sr
                q_sa[a] += (p * (r + gamma * v[s_]))
        policy[s] = np.argmax(q_sa)
    return policy


def value_iteration(env, gamma, epsilon=1e-20, max_iterations=100000):
    """
    This function implements value iteration algorithm for the infinite
    horizon discounted MDPs. If the sup norm of v_k - v_{k-1} is less than
    epsilon or number of iterations reaches max_iterations, it should return
    the value function.
    """
    start = time.time()
    v = np.zeros(env.observation_space.n)  # initialize value-function
    ########################### Your Code Here ###########################
    # Hint: see implementation of extract_policy
```

```
    ########################### End of your code ########################
    end = time.time()
    print("Value iteration took {0} seconds.".format(end - start))
    return v

if __name__ == '__main__':
    env_name  = 'FrozenLake8x8-v1'
    for gamma in [.9, .95, .99, .9999, 1]:
        print("-"*10, "Gamma={0}".format(gamma) ,"-"*10)
        env = gym.make(env_name)
        env.seed(1111)
        optimal_v = value_iteration(env, gamma);
        policy = extract_policy(optimal_v, gamma)
        policy_score = evaluate_policy(env, policy, gamma, n=1000)
        print('Average score = ', policy_score)
```

Policy Iteration

```
def compute_policy_v(env, policy, gamma):
    """ Iteratively evaluate the value-function under policy.
    Alternatively, we could formulate a set of linear equations in iterms of v[s]
    and solve them to find the value function.
    """
    v = np.zeros(env.observation_space.n)
    eps = 1e-10
    while True:
        prev_v = np.copy(v)
        for s in range(env.observation_space.n):
            policy_a = policy[s]
            v[s] = sum([p * (r + gamma * prev_v[s_]) for p, s_, r, _ in env.P[s][policy_a]])
        if (np.sum((np.fabs(prev_v - v))) <= eps):
            # value converged
            break
    return v

def policy_iteration(env, gamma, max_iterations=100000):
    """
    This function implements policy iteration algorithm.
    """
    start = time.time()
    policy = np.random.choice(env.action_space.n, size=(env.observation_space.n))  # initialize a random policy
    ########################### Your Code Here ###########################









    ########################### End of your code ########################
    end = time.time()
    print("Policy iteration took {0} seconds.".format(end - start))
    return policy


if __name__ == '__main__':
    env_name  = 'FrozenLake8x8-v1'
    for gamma in [.9, .95, .99, .9999, 1]:
        print("-"*10, "Gamma={0}".format(gamma) ,"-"*10)
        env = gym.make(env_name)
        env.seed(1111)
        optimal_policy = policy_iteration(env, gamma=gamma)
```

```
scores = evaluate_policy(env, optimal_policy, gamma=gamma)
print('Average scores = ', np.mean(scores))
```

Questions

1. How many iterations did it take for the value iteration to converge? How about policy iteration?

Note your answer here.

2. How much time did it take for the value iteration to converge? How about the policy iteration?

Note your answer here.

3. Which algorithm is faster? Why?

Note your answer here.

4. How does the average score change as γ gets closer to 1? Why?

Note your answer here.