Stanford CME 241 - Midterm Exam

Tom Morvan - SUID: 06393077

February 12, 2020

1 Optimal Croaking on Lily pads

1. - **State space**: $\mathcal{S} = \{0, ..., n\}$- Represents the lily pad the frog is on.
   - **Action space**: $\mathcal{A} = \{A, B\}$ - Represents the sound croaked by the frog.
   - **Transition function**:
     By considering $T = \mathbb{N}$

$$\forall t \in T, \forall i \in \mathcal{S} \setminus \{0, n\}, \mathbb{P}(S_{t+1} = i - 1 | S_t = i, A_t = A) = \frac{i}{n}$$
$$\forall t \in T, \forall i \in \mathcal{S} \setminus \{0, n\}, \mathbb{P}(S_{t+1} = i + 1 | S_t = i, A_t = A) = \frac{n-i}{n}$$
$$\forall t \in T, \forall i \in \mathcal{S} \setminus \{0, n\}, \forall j \in \mathcal{S} \setminus \{i\}, \mathbb{P}(S_{t+1} = j | S_t = i, A_t = B) = \frac{1}{n}$$
$$\forall t \in T, \mathbb{P}(S_{t+1} = 0 | S_t = 0) = \mathbb{P}(S_{t+1} = n | S_t = n) = 1$$

   - **Reward function**:
     Define the random variable R_t representing the reward obtained at time t with:

$$\forall t \in T, \forall i \in \mathcal{S} \setminus \{0, n\}, \mathbb{P}(R_t = 0 | S_t = i) = 1$$
$$\forall t \in T, \mathbb{P}(R_t = -1 | S_t = 0) = 1$$
$$\forall t \in T, \mathbb{P}(R_t = 1 | S_t = n) = 1$$

     We define the state reward function $\mathcal{R}_s^A = \mathbb{E}[R_{t+1} | S_t = s, A_t = A]$. We have:

$$\forall i \in \mathcal{S} \setminus \{0, 1, n-1, n\}, \mathcal{R}_i^A = 0$$
$$\forall i \in \mathcal{S} \setminus \{0, n\}, \mathcal{R}_i^B = 0$$
$$\mathcal{R}_1^A = \frac{-1}{n}$$
$$\mathcal{R}_{n-1}^A = \frac{1}{n}$$
$$\mathcal{R}_0^A = \mathcal{R}_0^B = -1$$
$$\mathcal{R}_n^A = \mathcal{R}_n^B = 1$$

2. (View Appendix)

3.

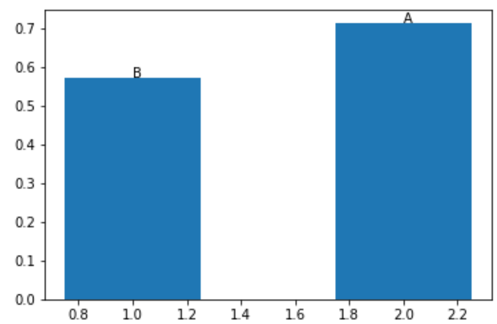

Figure 1: n = 3

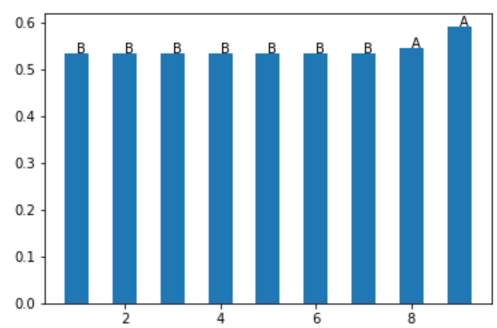

Figure 2: n = 10

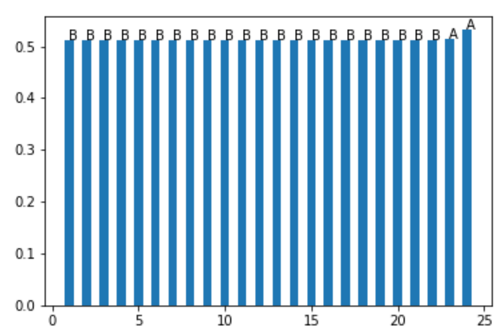

Figure 3: n = 25

We can see above the plotted Optimal Escape-Probability and of the associated Optimal Croak of every initial transient state for $n = 3$, $n = 10$ and $n = 25$. We remark that the optimal policy consists of croaking 'B' on all transient States except when we get close to the last lily pad $(n-2, n-1)$ where we croak 'A'.

2 Job-Hopping and Wage-Maximization

1.
   - **State space**: $\mathcal{S} = \{0, ..., n\}$- Represents the daily job, 0 being unemployed
   - **Action space**: $\mathcal{A} = \{A, D\}$ - A is accepting job offer, D declining job offer.
   - **Transition function**:
     By considering $T = \mathbb{N}$
     $\forall t \in T, \forall i \in \mathcal{S} \setminus \{0\}, \mathbb{P}(S_{t+1} = i | S_t = i, A_t = A) = \mathbb{P}(S_{t+1} = i | S_t = i, A_t = D) = 1 - \alpha$
     $\forall t \in T, \forall i \in \mathcal{S} \setminus \{0\}, \mathbb{P}(S_{t+1} = 0 | S_t = i, A_t = A) = \mathbb{P}(S_{t+1} = i | S_t = i, A_t = D) = \alpha$
     $\forall t \in T, \forall i \in \mathcal{S} \setminus \{0\}, \mathbb{P}(S_{t+1} = i | S_t = 0, A_t = A) = p_i$
     $\forall t \in T, \forall i \in \mathcal{S} \setminus \{0\}, \mathbb{P}(S_{t+1} = 0 | S_t = 0, A_t = D) = 1$
   - **Reward function**:
     Define the random variable R_t representing the reward obtained at time t with:
     $\forall t \in T, \forall i \in \mathcal{S}, \mathbb{P}(R_t = U(w_i) | S_t = i) = 1$

     We define the state reward function $\mathcal{R}_s^A = \mathbb{E}[R_{t+1} | S_t = s, A_t = A]$. We have:
     $\forall t \in T, \forall i \in \mathcal{S} \setminus \{0\}, \mathcal{R}_i^A = \mathcal{R}_i^D = (1 - \alpha)U(w_i) + \alpha U(w_0)$
     $\forall t \in T, \mathcal{R}_0^A = \sum_{i=1}^{n} p_i U(w_i)$
     $\forall t \in T, \mathcal{R}_0^D = U(w_0)$
   - **Bellman Optimality Equation**:

$$v_*(s) = \max_{a \in \mathcal{A}} (\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s'))$$

2. View Appendix

3 Solving a continuous states/actions MDP analytically

Let $s \in \mathbb{R}$ define our current state S_t for $t \in \mathbb{N}$.
We want to minimize the *Expected Discounted Sum of Costs* in a myoptic case, which is equivalent to minimizing the next state's (S_{t+1}) cost. Analytically, we can rewrite the problem:

$$\min_{a \in \mathbb{R}} \mathbb{E}[cost_{t+1} | S_t = s, A_t = a] \tag{1}$$

Which can be rewritten:

$$\min_{a \in \mathbb{R}} \mathbb{E}[e^{aS_{t+1}} | S_t = s, A_t = a] \tag{2}$$

We also know that $(S_{t+1} | S_t = s, A_t = a) \sim \mathcal{N}(s, \sigma^2)$

Therefore, (2) gives us:

$$\min_{a \in \mathbb{R}} e^{as + \frac{a^2 \sigma^2}{2}} \tag{3}$$

(3) is equivalent to:

$$\min_{a \in \mathbb{R}} (as + \frac{a^2 \sigma^2}{2}) \tag{4}$$

Which finally gives us:

$$a = -\frac{s}{\sigma^2}$$

And:

$$cost = e^{-\frac{s^2}{2\sigma^2}}$$

Conclusion: For any state $S_t = s \in \mathbb{R}$, the optimal action is $A_t = -\frac{s}{\sigma^2}$ with associated cost $e^{-\frac{s^2}{2\sigma^2}}$.

Question 1 Part 2

February 11, 2020

[1]:
```
from IPython.display import Image
import matplotlib.pyplot as plt
import numpy as np
from numpy.linalg import inv
```

[2]:
```
def get_states(n):
    return (range(0,n+1))
```

[3]:
```
def get_transition_A(n):
    P = np.zeros((n+1,n+1))
    P[0,0] = 1
    P[n,n] = 1
    for i in range(1,n):
        P[i,i+1] = (n-i)/n
        P[i,i-1] = i/n
    return(P)
```

[4]:
```
def get_transition_B(n):
    P = np.zeros((n+1,n+1))
    P[0,0] = 1
    P[n,n] = 1
    for i in range(1,n):
        for j in range(0,n+1):
            if i != j:
                P[i,j] = 1/n
    return(P)
```

[5]:
```
def get_rewards_A(n):
    R = np.zeros(n+1)
    R[0] = -1
    R[1] = -1/n
    R[n-1] = 1/n
    R[n] = 1
    return(R)
```

[6]:
```
def get_rewards_B(n):
    R = np.zeros(n+1)
    R[0] = -1
    R[n] = 1
```

```python
    return(R)
```

[7]:
```python
def update_state_val(n, gamma, v_old, transition_A, transition_B, rewards_A,␣
↪rewards_B):
    v_new = np.array([max(rewards_A[i] + gamma*np.dot(transition_A[i,:],v_old),
                          rewards_B[i] + gamma*np.dot(transition_B[i,:],v_old))␣
↪for i in range(0,n+1)])
    return(v_new)
```

[8]:
```python
def value_iteration(n_iter, n, gamma, v_0, transition_A, transition_B,␣
↪rewards_A, rewards_B):
    for k in range(0,n_iter):
        v_0 = update_state_val(n, gamma, v_0, transition_A, transition_B,␣
↪rewards_A, rewards_B)
    return(v_0)
```

[9]:
```python
def get_policy(n, gamma, v, transition_A, transition_B, rewards_A, rewards_B):
    policy = []
    for i in range(0,n+1):
        q_A = rewards_A[i] + gamma*np.dot(transition_A[i,:],v)
        q_B = rewards_B[i] + gamma*np.dot(transition_B[i,:],v)
        policy.append('A') if q_A > q_B else policy.append('B')
    return policy
```

[10]:
```python
def get_transition_policy(n, transition_A, transition_B, policy):
    transition_policy = np.zeros((n+1, n+1))
    for i in range(0,n+1):
        transition_policy[i,:] = transition_A[i,:] if policy[i] == 'A' else␣
↪transition_B[i,:]
    return(transition_policy)
```

[11]:
```python
def probability_distribution(n, transition_policy):

    #We reorder our transition matrix to
    #isolate transient states from absorbant states
    transition_policy = np.copy(transition_policy)
    transition_policy = np.roll(transition_policy,-1,axis=0)
    transition_policy = np.roll(transition_policy,-1,axis=1)

    Q = transition_policy[0:-2,0:-2]
    R = transition_policy[0:-2,-2:]
    N = inv(np.eye(n-1)-Q)

    result = np.dot(N,R)

    return(result[:,0])
```

[12]:
```python
def plot_result(n, result, policy):
        X = range(1,n)
        Y = result
        plt.figure()
        plt.bar(X,Y, width=0.5)
        for i in range(1, n):
            plt.annotate(policy[i], xy=(X[i-1],Y[i-1]))
        fig_name = 'figure' + str(n) + '.png'
        plt.savefig(fig_name)
        plt.show()
```

[13]:
```python
def MDP(n, gamma, n_iter):

    # ---- Building MDP ----
    S = get_states(n)
    transition_A = get_transition_A(n)
    transition_B = get_transition_B(n)
    rewards_A = get_rewards_A(n)
    rewards_B = get_rewards_B(n)
    # ----------------------

    # ---- Solving MDP -----
    v_0 = np.zeros(n+1)
    v_0[0] = -1
    v_0[n] = 1

    v = value_iteration(n_iter, n, gamma, v_0, transition_A, transition_B,␣
 ↪rewards_A, rewards_B)
    policy = get_policy(n, gamma, v, transition_A, transition_B, rewards_A,␣
 ↪rewards_B)
    transition_policy = get_transition_policy(n, transition_A, transition_B,␣
 ↪policy)
    # ----------------------

    # ---- Plot results -----
    result = probability_distribution(n, transition_policy)
    #plot_result(n, result, policy) - Commented for PDF export
    # -----------------------
```

[14]:
```python
MDP(3, 0.8, 1000)
```

[15]:
```python
MDP(10,0.8,1000)
```

[16]:
```python
MDP(25,0.8,1000)
```

Question 2 part 2

February 11, 2020

[1]:
```
import numpy as np
```

[2]:
```
def get_transition_A(n, P, alpha):
    transition_A = np.zeros((n+1,n+1))
    for i in range(1,n+1):
        transition_A[i,i] = 1 - alpha
        transition_A[i,0] = alpha
        transition_A[0,i] = P[i-1]
    return(transition_A)
```

[3]:
```
def get_transition_D(n, alpha):
    transition_D = np.zeros((n+1,n+1))
    transition_D[0,0] = 1
    for i in range(1,n+1):
        transition_D[i,i] = 1 - alpha
        transition_D[i,0] = alpha
    return(transition_D)
```

[4]:
```
def CRRA(x,a):
    if a==1:
        return(np.log(x))
    else:
        return((x**(1-a)-1)/(1-a))
```

[5]:
```
def get_rewards_A(n, W, P, alpha, a):
    R = np.zeros(n+1)
    for w, p in zip(W[1:],P):
        R[0] += p*CRRA(w, a)
    for i in range(1,n+1):
        R[i] = (1-alpha)*CRRA(W[i], a) + alpha*CRRA(W[0], a)
    return(R)
```

[6]:
```
def get_rewards_D(n, W, alpha, a):
    R = np.zeros(n+1)
    R[0] = CRRA(W[0], a)
    for i in range(1,n+1):
        R[i] = (1-alpha)*CRRA(W[i], a) + alpha*CRRA(W[0], a)
    return(R)
```

[7]:
```
def update_state_val(n, gamma, v_old, transition_A, transition_D, rewards_A,␣
↪rewards_D):
    v_new = np.array([max(rewards_A[i] + gamma*np.dot(transition_A[i,:],v_old),
                          rewards_D[i] + gamma*np.dot(transition_D[i,:],v_old))␣
↪for i in range(0,n+1)])
    return(v_new)
```

[8]:
```
def value_iteration(n_iter, n, gamma, v_0, transition_A, transition_D,␣
↪rewards_A, rewards_D):
    for k in range(0,n_iter):
        v_0 = update_state_val(n, gamma, v_0, transition_A, transition_D,␣
↪rewards_A, rewards_D)
    return(v_0)
```

[9]:
```
def get_policy(n, gamma, v, transition_A, transition_D, rewards_A, rewards_D):
    policy = []
    for i in range(0,n+1):
        q_A = rewards_A[i] + gamma*np.dot(transition_A[i,:],v)
        q_D = rewards_D[i] + gamma*np.dot(transition_D[i,:],v)
        policy.append('A') if q_A > q_D else policy.append('D')
    return policy
```

[10]:
```
# ---- Input -----
n = 10
S = range(0,n+1)
W = range(1,n+2)
alpha = 0.1
a = 0.5
P = [1/n for i in range(0,n)]
n_iter = 1000
gamma = 0.5

# ---- Building MDP ----
transition_A = get_transition_A(n, P, alpha)
transition_D = get_transition_D(n, alpha)
rewards_A = get_rewards_A(n, W, P, alpha, a)
rewards_D = get_rewards_D(n, W, alpha, a)
# ----------------------

# ---- Solving MDP -----
v_0 = np.ones(n+1) #Random init

v = value_iteration(n_iter, n, gamma, v_0, transition_A, transition_D,␣
↪rewards_A, rewards_D)
policy = get_policy(n, gamma, v, transition_A, transition_D, rewards_A,␣
↪rewards_D)
```

[11]:
```
policy
```

[11]: ['A', 'D', 'D', 'D', 'D', 'D', 'D', 'D', 'D', 'D', 'D']