COMP 767 Reinforcement Learning
Assignment 1

Ali Rezagholizadeh

January 30, 2019

1 Question 1(b)

In this paper, authors tried to find problem-independent and problem-dependent boundary for the *finite time expected regret* of Thompson Sampling. we ignore describing Thompson Sampling method in this assignment because of being out of the scope of requirements in this question.

1.1 Summary of results

While several works had been done on finding boundary for *expected regret* in the bandit problem like *Lai and Robins* (who proposed the lower bound of any bandit problem eq.1) [15] [1], *Auer et al.* (which proposed upper bound for UCB1 method eq.2) [4], and *Kaufmann et al.* [13 and 14], the authors of this paper achieved a problem-dependent upper boundary (eq.3) which approaches asymptotic lower bound of *Lai and Robbins* and match those provided by recent work for Thompson Sampling [14].

They also achieved a problem-independent upper bound (eq. 4) which approaches problem-independent lower bound of $\Omega(\sqrt{NT})$ which achieved by Bubeck et al. [5] for this problem.

$$E[R(T)] \geqslant \left[\sum_{i=2}^{N} \frac{\Delta_i}{d(\mu_i, \mu_1)} + o(1)\right] \ln T \tag{1}$$

$$E[R(T)] \leqslant \left[8 \sum_{i=2}^{N} \frac{1}{\Delta_i}\right] lnT + (1 + \pi^2/3) \left(\sum_{i=2}^{N} \Delta_i\right) \tag{2}$$

$$E[R(T)] \leqslant (1+\epsilon) \sum_{i=2}^{N} \frac{lnT}{d(\mu_i, \mu_1)} \Delta_i + O(\frac{N}{\epsilon^2}) \tag{3}$$

$$E[R(T)] \leqslant O(\sqrt{NTlnT}) \tag{4}$$

These two problem-dependent and problem-independent bounds (eq.3 and 4) were brought as Theorems in the paper.

1.2 Main steps and ideas in the proof

Authors constrained their work to have Thompson Sampling with Bernoulli reward (reward of 0 or 1), but as they confirmed this can be extended to general reward distribution using [1]. This limitation to have Bernoulli reward helps authors to get a benefit from conjugate of Bernoulli, Beta distribution, as prior distribution ($p(R)$) of the reward. This makes posterior ($p(R \mid h)$, where h is: $a1, r1, a2, r2, .., a_t, r_t$) also being Beta distribution.
Posterior of each arm changes whenever the arm is played. Whatever the

[1] This is the reference number used in the paper.

reward achieved by playing an arm be one (success) (or zero (fail)), then the mean of its posterior distribution (expected reward according to this current distribution) approaches to one (or zero) and it becomes tighter at the mean point.

Sampling from these distributions and then selecting the next arm to play according to the value of each sample, makes possible for us to estimate the expected regret (or at least find a boundary for it) in time T. The authors did this by getting benefit from these distributions.

In other side, the authors took beautiful strategy by considering two arbitrary points, x_i and y_i for each arm so that $\mu_i < x_i < y_i < \mu_1$, where μ_i is the value function of arm $i \neq 1$ (here, it should be stated that the authors, without loss of generality, assumed that the first arm is the unique optimal arm, i.e., $\mu^* = \mu_1 > argmax_{i \neq 1} \mu_i$).

By considering of these x_i and y_i, they constructed the foundation of their inequalities used in the steps (lemmas) throughout the way of proving the Theorems.

The concept behind using x_i and y_i could be both:

1. Give the opportunity to design the boundary whatever they want in the share point:

$$E[K_i(T)] \leqslant \frac{24}{\Delta_i'^2} + \sum_{j=0}^{T-1} \Theta\left(e^{-\Delta_i'^2 j/2} + \frac{1}{(j+1)\Delta_i'^2} e^{-D_i j} + \frac{1}{e^{\Delta_i'^2 j/4} - 1} \right) + L_i(T) + 1 + \frac{1}{d(x_i, \mu_i)} + 1. \tag{5}$$

   , where the way of obtaining problem-dependent bound and problem-independent bound are separating.

2. assure that after a time t (before T), so that $for\ \ every \quad i \neq 1$, $K_i(t) > L_i(T)$, there exist such inequality in the aforementioned share point. This because $L_i(T)$ is the function of T, x_i, and y_i.

The main steps through proving theorems are four lemmas whose aim is to prove the share equation (eq. 5) and then by selecting suitable X_i and Y_i (settings in (6)) they reached to the mentioned theorems. It is worth mentioning that, through the way, author got advantages of Chernaff-Hoeffiding bound and 'Beta-Binomial trick' facts a lot.

$$\begin{cases} d(x_i, \mu_1) = d(\mu_i, \mu_1)/(1+\epsilon) \ \textbf{and} & \\ d(x_i, y_i) = d(x_i, \mu_1)/(1+\epsilon) = d(\mu_i, \mu_1)/(1+\epsilon)^2 & \text{to reach problem-dependent bound,} \\ & \\ x_i = \mu_i + \frac{\Delta_i}{3} \ \textbf{and} \ \ y_i = \mu_1 - \frac{\Delta_i}{3} & \text{to reach problem-independent bound.} \end{cases} \tag{6}$$

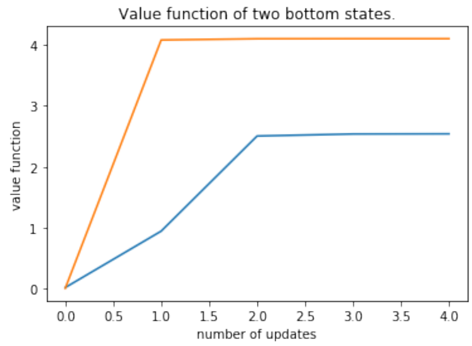
2 Question 2(a)

I could implement Policy iteration completely. I bring the result of my code here and my raw code after:

1. p = 0.7 and n =5: Total update in policy iteration: 4 Final greedy policy and its value-function:

```
| term |    R |    R |    R | term |
|    R |    R |    R |    R |    U |
|    R |    R |    R |    U |    U |
|    R |    R |    U |    U |    U |
|    R |    U |    U |    U |    U |
```

```
| 0 | 5.2405887925621055 | 6.875761363794614 | 8.723676358173222 | 0 |
| 4.156965591764831 | 5.478203156223082 | 6.477986299407242 | 7.549084423452747 | 8.724719871571663 |
| 3.8682767418740753 | 4.82266935665731 | 5.609188915350512 | 6.481219238604924 | 6.891213164547389 |
| 3.3199941309863172 | 4.1399674406401745 | 4.826129030029215 | 5.510483427105133 | 5.455393907238159 |
| 2.540626167075174 | 3.3214355869812446 | 3.886252443697519 | 4.382699323992172 | 4.100969454668087 |
```

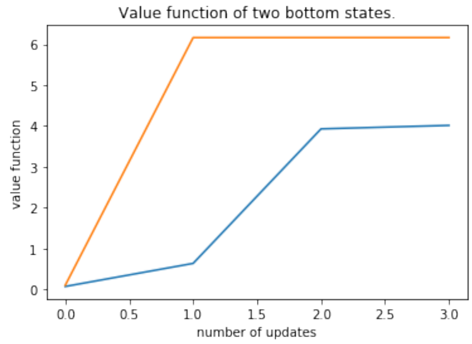


2. p = 0.9 and n =5:
   Total update in policy iteration: 3 Final greedy policy:

```
| term |    R |    R |    R | term |
|    R |    R |    R |    R |    U |
|    R |    R |    R |    U |    U |
|    R |    R |    U |    U |    U |
|    R |    U |    U |    U |    U |
```

```
| 0 | 7.128892329611076 | 8.349886293709046 | 9.630877789108 | 0 |
| 5.790348385629835 | 6.7827685259004475 | 7.634687771942954 | 8.578015472911629 | 9.63093775009695 |
| 5.275605122835685 | 6.0537192831497935 | 6.800231175069826 | 7.634869923265149 | 8.352551201646104 |
| 4.693923689708901 | 5.387312419175714 | 6.0539062034463065 | 6.78819738182139 | 7.244931180934402 |
| 4.01335752282207 | 4.694002915325792 | 5.278436309672564 | 5.908637001806977 | 6.164349540668548 |
```

3. p = 0.7 and n =50:
   It's run took a lot of time...



2.1 Code:

```
import numpy as np

def Setting_Rewards():

    Rewards = [0, 1, 10]

    return Rewards

def Setting_Actions():
    Actions = ["U", "D", "L", "R"]

    return Actions


def Convert_position_list_to_str(s):
    # s is [,] position.
    s_str = '['
    s_str += ''.join(str(x) for x in s)
    s_str += ']'

    return s_str
def Convert_position_str_to_list(s_str):

    r = int(s_str[1])
    c = int(s_str[2])

    s= [r,c]

    return s

def greedy_policy_of_current_policy(S_b, S_t, V):
    import random
    Greedy_Policy = {}
    for s in S_b:
        Action_set = Deterministic_Actions(s,V)
        if(Action_set == "Error"):
            print("Error occurred in obtaining greedy policy in state: ", s)
```

```
        else:
            s_str = Convert_position_list_to_str(s)
            if(len(Action_set) > 0):
                n_a = len(Action_set)
                direction_n = int(random.uniform(0,n_a))
                direction = Action_set[direction_n]
                d = {s_str: direction}
                Greedy_Policy.update(d)

    for s in S_t:
        s_str = Convert_position_list_to_str(s)
        d = {s_str: ""}
        Greedy_Policy.update(d)



    # Greedy_Policy  is a dict  like: {"[00]":'U'}
    return Greedy_Policy


def Determine_States(n):
    # This is determined according the requirements of the ass1_q2(a) in Comp 767.
    # the states are position like [[,][,]...]
    S_t = [[0,0] , [0,n-1]]
    # S_t is terminal state.
    S_b = [[i,j] for i in range(n) for j in range(n)]
    S_b.pop(n-1)
    S_b.pop(0)
    # S_b is in between _ not terminal state.

    return S_t, S_b


def Inital_value_function(S_b,S_t):
    # S_t  is terminal states,  and 'S_b'  is between sates. Each is list of positions. [[,] [,] ..]
    import math
    n1 = len(S_b)
    n2 = len(S_t)

    n = n1+n2
    width = int(math.sqrt(n))
    Value_Function = []
    for i in range(width):
        r = [0 for j in range(width)]
        Value_Function.append(r)

    return Value_Function


def Deterministic_Actions(s,V):
    # this is deterministic action selection according to policy Pai.
```

```
    # So, it select an action according to value function of that policy.
    # input: s is a position of one point, list of two numbers (position in gride- which is starting from 0)     -
    #         ... V is Value function of a policy (is matrix of gride size).
    import numpy as np
    r = s[0]
    c = s[1]

    All_logical_neighbors = [[r-1, c], [r+1 , c], [r, c-1] , [r, c+1]]
    end_of_grid = len(V)
    Possible_neighbors = [z for z in All_logical_neighbors if(z[0]>-1 and z[0]< end_of_grid and z[1]>-1 and z[1]< end_of_grid)]
    max_neighbor_value = -10

    for position in Possible_neighbors:
        value = V[position[0]][position[1]]
        if(max_neighbor_value < value):
            max_neighbor_value = value

    max_neighbor_pos = []
    for position in Possible_neighbors:
        value = V[position[0]][position[1]]
        if(max_neighbor_value == value):
            max_neighbor_pos.append(position)
    # all neighbors who have the max value would enter to 'max_neighbor_pos'.

    action_set = ""
    s_array = np.array(s)
    for max_Neigh in max_neighbor_pos:
        max_neighbor_arr = np.array(max_Neigh)
        move = max_neighbor_arr - s_array

        if(move[0] == -1):
            action_set += "U"
        elif(move[0] == 1):
            action_set += "D"
        else:
            if(move[0] != 0):
                print("Error in Choosing_deterministic_Action, 1.", move[0])
                action_set = "Error"


        if(move[1] == -1):
            action_set += "L"
        elif(move[1] == 1):
            action_set += "R"
        else:
            if(move[1] != 0):
```

```
                print("Error in Choosing_deterministic_Action, 2." , move[1])
                action_set = "Error"



    return action_set


def Printing_ValueFunction(matrix_list):
    # 'matrix_list' is [[,,..][,,..]...]
    r_len = len(matrix_list)
    c_len = len(matrix_list[0])
    print("Value Function:", end ='\n')
    for i in range(r_len):
        for j in range(c_len):
            if(j==0):
                print(end=' | ')
                print(matrix_list[i][j], end=' | ')
            else:
                print(matrix_list[i][j], end=' | ')
        print('')
    return

def Printing_Deterministic_Policy(g_p,S_t, S_b):
    # g_p is dict of like {'[11]': "UDLR"}
    import math
    n1 = len(S_b)
    n2 = len(S_t)

    n = n1+n2
    width = int(math.sqrt(n))

    Greedy_Policy_mat = []
    for i in range(width):
        r = ['' for j in range(width)]
        Greedy_Policy_mat.append(r)

    for s in S_b:
        s_str = Convert_position_list_to_str(s)
        Greedy_Policy_mat[s[0]][s[1]] = g_p[s_str]


    for s in S_t:
        #s_str = Convert_position_list_to_str(s)
        Greedy_Policy_mat[s[0]][s[1]] = "term"


    # printing
    matrix_list = Greedy_Policy_mat
    r_len = len(matrix_list)
    c_len = len(matrix_list[0])

    #'value %3s - num of occurances = %d' % item
```

```python
    #'{0:10} ==> {1:10d}'.format(name, phone)


    for i in range(r_len):
        for j in range(c_len):
            if(j==0):
                print(end=' | ')
                print("%5s |"% matrix_list[i][j], end='')
            else:
                print("%5s |"% matrix_list[i][j], end='')
        print('')


    return



# ==========================================================================================
# DP part:
#


def Policy_Iteration(S_b, S_t, Rewards, Actions, Policy, V, Discount, P_Dynamic, e):
    import math
    Greedy_Policy = Policy
    # Greedy_Policy  is dict of like {'[12]':'U'}
    Theta = 0.001

    S_p = S_b + S_t
    n = len(S_p)
    n = int(math.sqrt(n))
    policy_stable = False
    # S_p is s+
    Number_of_update_policy = 1
    V_S_Left = []
    V_S_Right = []
    while(policy_stable == False):

        print("=================================== update: ", Number_of_update_policy," =================================")
        print("Greedy policy of current policy: \n")
        Printing_Deterministic_Policy(Greedy_Policy,S_t, S_b)

        print("Computing V_i through the value function of aforementioned greedy policy ... ... .")
        Delta = 2
        while(Theta < Delta):
            Delta = 0
            for s in S_b:
                # s is [r,c] as the position of current state.
```

```
                r_s = s[0]
                c_s = s[1]
                current_v = V[r_s][c_s]
                s_str = Convert_position_list_to_str(s)
                greedy_action = Greedy_Policy[s_str]

                Greedy_Policy_ = E_greedy_policy(Greedy_Policy, S_b, Actions, e)
                actions, p_ = Greedy_Policy_[s_str]
                if(greedy_action != ""):
                    # when s is not terminal
                    sum__ = 0
                    for a_i in range(len(Actions)):
                        pi = p_[a_i]
                        ac = Actions[a_i]
                        sum_ = 0
                        for s_ in S_p:
                            for r in Rewards:
                                p_dy = Dynamic_p(s_,r,s,ac, P_Dynamic)
                                if(p_dy != 0):
                                    #print("s, greedy_action, S_, r, p",s, greedy_action, s_, r , p_dy)
                                    sum_ += p_dy*(r + Discount*V[s_[0]][s_[1]])
                        sum__ += pi * sum_

                    new_v = sum__
                    V[r_s][c_s] = new_v
                    diff = abs(current_v - new_v)
                    #print("diff: ", diff, end=' - ')
                    Delta = max(Delta , diff)
        #print()
        #print("Theta and Delta: ",Theta , Delta, end=' - ')
    print("Value_function obtained from 'Iterative policy evaluation' is: ")
    Printing_ValueFunction(V)
    V_S_Left.append(V[n-1][0])
    V_S_Right.append(V[n-1][n-1])
    print("-----------------------------------")
    print("Now updating greedy policy using this Value_function achieved ... ..")
    policy_stable = True
    for s in S_b:
        s_str = Convert_position_list_to_str(s)
        current_action = Greedy_Policy[s_str]

        max_action_value = 0
        max_q_action = ""
        for a in Actions:
            sum_ = 0
            for s_ in S_p:
                for r in Rewards:
                    p_dy = Dynamic_p(s_,r,s,a, P_Dynamic)
```

```
                        if(p_dy != 0):
                            sum_ += p_dy*(r + Discount* V[s_[0]][s_[1]])

                if(max_action_value <= sum_):
                    max_action_value = sum_
                    max_q_action = a


            Greedy_Policy[s_str] = max_q_action

            if(current_action != max_q_action):
                policy_stable = False

        print("Greedy policy of updated policy: \n")
        Printing_Deterministic_Policy(Greedy_Policy,S_t, S_b)
        if(policy_stable == False):
            Number_of_update_policy += 1
            print("Upadted policy is not stable so we go through obtaining its Value_fuction, then updating g_policy again. ")
            print("==================================================================================")
            print("||||||||||||||||||||||||||||||||||||||")
        else:
            print("No update was needed. The last one is the optimal policy.")




    return V_S_Left, V_S_Right, Number_of_update_policy


#----------------------------------------------------------------------------------
def E_greedy_policy(Greedy_Policy, S_b, Actions, e):
    Greedy_Policy_ = {}
    for s in S_b:
        s_str = Convert_position_list_to_str(s)
        greedy_action = Greedy_Policy[s_str]
        indx = 0
        index = -1
        for ac in Actions:
            if(ac == greedy_action):
                index = indx
            indx += 1
        p = [0 for i in range(len(Actions))]
        p[index] = e
        e_ = (1-e)/4
        p_ = [i + e_ for i in p]
        r = {s_str: [Actions, p_]}
```

```
        Greedy_Policy_.update(r)

    return Greedy_Policy_
def E_greedy(Greedy_Policy, S_b, Actions, e):
    import random

    for s in S_b:
        s_str = Convert_position_list_to_str(s)
        greedy_action = Greedy_Policy[s_str]

        t = random.uniform(0,1)
        if(t<e):
            selection = greedy_action
        else:
            t_2 = random.uniform(0,1)
            if(t_2 < 0.25):
                selection = Actions[0]
            elif(t_2 < 0.5):
                selection = Actions[1]
            elif(t_2 < 0.75):
                selection = Actions[2]
            else:
                selection = Actions[3]

    return selection

# -----------------------------------------------------------------------------------------------------
# Regarding Dynamic:
def Possible_Action_for_states(S_b,S_t):
    import numpy as np
    Possible_Actions_for_states = {}
    terminal_1_str = Convert_position_list_to_str(S_t[0])
    terminal_2_str = Convert_position_list_to_str(S_t[1])
    for s in S_b:
        r = s[0]
        c = s[1]

        All_logical_neighbors = [[r-1, c], [r+1 , c], [r, c-1] , [r, c+1]]
        end_of_grid = len(V)
        Possible_neighbors = [z for z in All_logical_neighbors if(z[0]>-1 and z[0]< end_of_grid and z[1]>-1 and z[1]< end_of_grid)]
        max_neighbor_value = -10

        s_array = np.array(s)
        action_set = ""
        Next_state_set =[]
        Rewards = []
        s_array = np.array(s)
        for pos_N in Possible_neighbors:
            pos_neighbor_arr = np.array(pos_N)
```

```
            move = pos_neighbor_arr - s_array

            if(move[0] == -1):
                action_set += "U"
                N_state_str = Convert_position_list_to_str(pos_N
)
                Next_state_set.append(N_state_str)
                if(N_state_str == terminal_1_str):
                    Rewards.append(1)
                elif(N_state_str == terminal_2_str):
                    Rewards.append(10)
                else:
                    Rewards.append(0)
            elif(move[0] == 1):
                action_set += "D"
                N_state_str = Convert_position_list_to_str(pos_N
)
                Next_state_set.append(N_state_str)
                if(N_state_str == terminal_1_str):
                    Rewards.append(1)
                elif(N_state_str == terminal_2_str):
                    Rewards.append(10)
                else:
                    Rewards.append(0)
            else:
                if(move[0] != 0):
                    print("Error in
Choosing_deterministic_Action, 1.", move[0])
                    action_set = "Error"


            if(move[1] == -1):
                action_set += "L"
                N_state_str = Convert_position_list_to_str(pos_N
)
                Next_state_set.append(N_state_str)
                if(N_state_str == terminal_1_str):
                    Rewards.append(1)
                elif(N_state_str == terminal_2_str):
                    Rewards.append(10)
                else:
                    Rewards.append(0)
            elif(move[1] == 1):
                action_set += "R"
                N_state_str = Convert_position_list_to_str(pos_N
)
                Next_state_set.append(N_state_str)
                if(N_state_str == terminal_1_str):
                    Rewards.append(1)
                elif(N_state_str == terminal_2_str):
                    Rewards.append(10)
                else:
                    Rewards.append(0)
            else:
```

```python
                if(move[1] != 0):
                    print("Error in Choosing_deterministic_Action , 2." , move[1])
                    action_set = "Error"

        s_str = Convert_position_list_to_str(s)
        state_action = {s_str:[action_set , Next_state_set , Rewards]}
        Possible_Actions_for_states.update(state_action)
    for s in S_t:
        action_set = ""
        Next_state_set = []
        Rewards = []
        s_str = Convert_position_list_to_str(s)
        state_action = {s_str:[action_set , Next_state_set , Rewards]}
        Possible_Actions_for_states.update(state_action)
    return Possible_Actions_for_states

def Dynamic_p(s_,r,s,a , P):
    # s is like [1,1] ; a is like "U" ; r is like 0 ; s_ is like [3,3]
    # P is like: {'[01]':[Actions, Next_state, Rewards]}
    # S_b
    s_str = Convert_position_list_to_str(s)
    Actions, Next_state, Rewards = P[s_str]
    #print(Actions)
    indx = 0
    index = -1
    for ac in Actions:
        if(ac == a):
            index = indx
        indx += 1

    # now index is corresponding action to take, next state, next reward considering this action.
    if(index == -1):
        return 0
    s__str = Convert_position_list_to_str(s_)
    if(Next_state[index] != s__str):
        return 0
    if(Rewards[index] != r):
        return 0

    return 1

    # for each 'current state', 'action' and 'reward' , 'new state' it should return the dynamic of
#-------------------------------------------------------------------------------


def Value_Iteration():

```

```python
    return

def Modified_Policy_Iteration():

    return

--------------------------------------------------------------------------------

def Main():
    S_t, S_b = Determine_States(50)
    Rewards = Setting_Rewards()
    Actions = Setting_Actions()
    V = Inital_value_function(S_b,S_t)
    e = 0.7
    Discount = 0.9
    g_Policy = greedy_policy_of_current_policy(S_b, S_t, V)
    P = Possible_Action_for_states(S_b,S_t)  # this is created for specific question
    V_S_Left, V_S_Right, Number_of_update_policy = Policy_Iteration(S_b, S_t, Rewards, Actions, g_Policy, V, Discount, P, e)

    import matplotlib.pyplot as plt

    # x axis values
    x = [i for i in range(Number_of_update_policy)]
    # corresponding y axis values
    y_1 = V_S_Left
    # plotting the points
    plt.plot(x, y_1, label = "Botton left")

    y_2 = V_S_Right
    plt.plot(x, y_2, label = "Botton Right")
    # naming the x axis
    plt.xlabel('number of updates')
    # naming the y axis
    plt.ylabel('value function')

    # giving a title to my graph
    plt.title('Value function of two bottom states.')

    # function to show the plot
    plt.show()

    return
```

Listing 1: My code