Question 1

Note : In all the questions, the X player is the random player and the O player is the agent / MDP player.

States = {'x__', 'x__', 'xox__', ...} - i.e. each state in the state set is the current board state after the X player has played its turn

Each state is represented as a string capturing the current Tic-Tac-Toe board state.

Actions = {0, 1, 2, 3, 4, 5, 6, 7, 8} - i.e. placing an X tile on any one of the open spots on the board

In [1]:
```
def print_grid(state):
    st = list(state)
    print(' {:1} | {:1} | {:1}'.format(state[0],state[1],state[2]))
    print('-----------')
    print(' {:1} | {:1} | {:1}'.format(state[3],state[4],state[5]))
    print('-----------')
    print(' {:1} | {:1} | {:1}'.format(state[6],state[7],state[8]))
    print('\n')



print('5 Terminal states where R(s) = 10')

print_grid('xox_ox_o_')
print_grid('xoxxoo_ox')
print_grid('x_xooox__')
print_grid('xxo_oxo__')
print_grid('xxo_xo__o')
```

```
5 Terminal states where R(s) = 10
 x | o | x
-----------
 _ | o | x
-----------
 _ | o | _


 x | o | x
-----------
 x | o | o
-----------
 _ | o | x


 x | _ | x
-----------
 o | o | o
-----------
 x | _ | _


 x | x | o
-----------
 _ | o | x
-----------
 o | _ | _


 x | x | o
-----------
 _ | x | o
-----------
```

```
_ | _ | o
```

In [2]:

```
import numpy as np
import copy
from collections import defaultdict

WINNING_POSITIONS = [(0,1,2), (3,4,5), (6,7,8), (0,3,6), (1,4,7), (2,5,8), (0,4,8),(2,4,6)]
gamma = 0.9
X_PLAYER, O_PLAYER = 1, 2
total_states = 0

def check_win(state):
    for a, b, c in WINNING_POSITIONS:
        if state[a] == state[b] == state[c] and state[a] == 'x':
            return -10
        elif state[a] == state[b] == state[c] and state[a] == 'o':
            return 10

    # draw
    if len(valid_moves(state)) == 0:
        return 0
    # non terminal state
    else:
        return 1

def valid_moves(state):
    s = np.array(list(state))
    return np.asarray(np.where(s == '_')).flatten()


initial_state = '_________'
states = defaultdict()
root_states = []


def generate_child_states(parent):
    #print(list(parent))

    available_moves = valid_moves(parent)

    for move in available_moves:
        s = list(parent)
        s[move] = 'o'
        s = ''.join(s)
        #print(s)

        reward = check_win(s)
        if reward != 1:
            states[s] = reward
            continue

        new_available_moves = valid_moves(s)
        for position in new_available_moves:
            new_parent = list(s)
            new_parent[position] = 'x'
            new_parent = ''.join(new_parent)
            #print(new_parent)

            r = check_win(new_parent)

            if r == 1:
                generate_child_states(new_parent)
```

```
            states[new_parent] = r
    return


# Generate roots
def generate_initial_states():

    for i in range(9):
        state = list(initial_state)
        state[i] = 'x'
        state = ''.join(state)
        root_states.append(state)

        states[state] = check_win(state)

        # Generate child states
        generate_child_states(state)


generate_initial_states()
```

In [3]:

```
states_keys = list(states.keys())
probability_table = np.zeros((9, len(states), len(states)))
rewards_list = np.zeros(len(states))
```

In [4]:

```
def transition_probabilities(parent):
    available_moves = valid_moves(parent)
    parent_index = states_keys.index(parent)

    for move in available_moves:
        s = list(parent)
        s[move] = 'o'
        s = ''.join(s)

        if check_win(s) != 1:
            rewards_list[states_keys.index(s)] = states[s]
            probability_table[move, parent_index, states_keys.index(s)] = 1 # double check this
            continue

        for position in valid_moves(s):
            new_parent = list(s)
            new_parent[position] = 'x'
            new_parent = ''.join(new_parent)

            probability_table[move, parent_index, states_keys.index(new_parent)] = 1/len(valid_moves(s)) # double check this
            rewards_list[states_keys.index(new_parent)] = states[new_parent]


            if check_win(new_parent) != 1:
                continue
            else:
                transition_probabilities(new_parent)
    return
```

In [5]:

```
for root in root_states:
    rewards_list[states_keys.index(root)] = states[root]
    transition_probabilities(root)
```

Question 2

In [6]:

```
# Printing trajectories

rtg_list = []

def print_trajectory(parent):

    print_grid(parent)

    available_moves = valid_moves(parent)
    choice = np.random.choice(available_moves)
    s = list(parent)
    s[choice] = 'o'
    s = ''.join(s)

    reward = check_win(s)
    if reward != 1:
        print_grid(s)
        rtg_list.append(states[s])
        return states[s]


    next_x_moves = valid_moves(s)
    choice = np.random.choice(next_x_moves)
    new_parent = list(s)
    new_parent[choice] = 'x'
    new_parent = ''.join(new_parent)

    r = check_win(new_parent)

    if r != 1:
        print_grid(new_parent)
        rtg_list.append(states[new_parent])
        return states[new_parent]
    else:
        rtg = states[new_parent] + gamma * print_trajectory(new_parent)
        rtg_list.append(rtg)
        return rtg



random_seeds = [0, 10, 20, 30, 40]
for i in range(len(random_seeds)):
    np.random.seed(random_seeds[i])
    print('Reward to go for state1 is ' + str(states[root_states[i]] + gamma * print_trajectory(root_states[i])))

    for j in range(len(rtg_list)):
        print('Reward to go for state' + str(j+2) + ' is ' + str(rtg_list[len(rtg_list) - j - 1]))

    print('-------------------------------------------------------------------')

    rtg_list = []
```

```
 x | _ | _
-----------
 _ | _ | _
-----------
 _ | _ | _


 x | _ | _
-----------
```

```
 _ | _ | o
-----------
 _ | x | _


 x | o | _
-----------
 _ | _ | o
-----------
 x | x | _


 x | o | _
-----------
 x | _ | o
-----------
 x | x | o


Reward to go for state1 is -4.58
Reward to go for state2 is -6.2
Reward to go for state3 is -8.0
Reward to go for state4 is -10
------------------------------------------------------------------
 _ | x | _
-----------
 _ | _ | _
-----------
 _ | _ | _


 _ | x | o
-----------
 _ | _ | _
-----------
 _ | x | _


 x | x | o
-----------
 _ | _ | _
-----------
 o | x | _


 x | x | o
-----------
 _ | o | _
-----------
 o | x | _


Reward to go for state1 is 10.0
Reward to go for state2 is 10.0
Reward to go for state3 is 10.0
Reward to go for state4 is 10
------------------------------------------------------------------
 _ | _ | x
-----------
 _ | _ | _
-----------
 _ | _ | _


 _ | _ | x
-----------
 x | o | _
-----------
 _ | _ | _
```

```
 _ | _ | x
-----------
 x | o | x
-----------
 _ | o | _


 _ | o | x
-----------
 x | o | x
-----------
 _ | o | _


Reward to go for state1 is 10.0
Reward to go for state2 is 10.0
Reward to go for state3 is 10.0
Reward to go for state4 is 10
------------------------------------------------------------------
 _ | _ | _
-----------
 x | _ | _
-----------
 _ | _ | _


 _ | _ | _
-----------
 x | _ | _
-----------
 o | x | _


 _ | _ | _
-----------
 x | _ | x
-----------
 o | x | o


 o | _ | _
-----------
 x | x | x
-----------
 o | x | o


Reward to go for state1 is -4.58
Reward to go for state2 is -6.2
Reward to go for state3 is -8.0
Reward to go for state4 is -10
------------------------------------------------------------------
 _ | _ | _
-----------
 _ | x | _
-----------
 _ | _ | _


 _ | _ | _
-----------
 x | x | _
-----------
 _ | o | _


 x | _ | _
```

```
-----------
 x | x | _
-----------
 _ | o | o


 x | o | _
-----------
 x | x | _
-----------
 x | o | o


Reward to go for state1 is -4.58
Reward to go for state2 is -6.2
Reward to go for state3 is -8.0
Reward to go for state4 is -10
-----------------------------------------------------------------------
```

Question 3

In [7]:

```
values = np.zeros(len(states))
actions = [0 for i in range(len(states))]


def check_for_convergence(old_vals):
    diff = values - old_vals

    if np.linalg.norm(abs(diff), np.inf) < 0.1:
        return True;
    else:
        return False;


while True:
    old_vals = copy.deepcopy(values)

    for i in range(len(states)):
        win = check_win(states_keys[i])
        if win != 1:
            values[i] = rewards_list[i]
            continue

        maximum = -1000
        action = 0

        for j in range(9):

            if np.sum(probability_table[j, i,:]) == 0:
                continue

            new_val = rewards_list[i] + gamma * (np.dot(probability_table[j, i, :], values))

            if new_val > maximum:
                maximum = new_val
                action = j

        values[i] = maximum
        actions[i] = action

    if check_for_convergence(old_vals) == False:
        continue;
    else:
        break;
```

In [8]:
```
print(set(values))
```

```
{0.0, 1.0, 1.9, 1.3, 4.0, 1.3000000000000003, 4.6, 7.3, 7.299999999999999, 9.5626, 10.0, 9.514000000000001, 7.9479999999999995, 4.438000000000001, 6.49, 8.920000000000002, 8.541999999999998, 8.434000000000001, 9.736171428571428, 4.923999999999995, 5.41, 6.004, 6.112, 7.948, 8.056000000000001, 8.312885714285715, 9.028, 9.562600000000002, -0.8, -1.3999999999999995, -0.7999999999999998, 4.924, 4.438, 5.409999999999999, 6.004000000000004, -10.0, -8.0, -7.399999999999999, -1.4000000000000004, -2.0, -1.4}
```

In [9]:
```
def print_value_trajectory(parent):

    print_grid(parent)
    print('Value of the above state is ' + str(values[states_keys.index(parent)]))

    available_moves = valid_moves(parent)
    choice = np.random.choice(available_moves)
    s = list(parent)
    s[choice] = 'o'
    s = ''.join(s)

    reward = check_win(s)
    if reward != 1:
        print_grid(s)
        print('Value of the above state is ' + str(values[states_keys.index(s)]))
        return states[s]


    next_x_moves = valid_moves(s)
    choice = np.random.choice(next_x_moves)
    new_parent = list(s)
    new_parent[choice] = 'x'
    new_parent = ''.join(new_parent)

    r = check_win(new_parent)

    if r != 1:
        print_grid(new_parent)
        print('Value of the above state is ' + str(values[states_keys.index(new_parent)]))
        return
    else:
        print_value_trajectory(new_parent)
        return



random_seeds = [0, 10, 20, 30, 40]
for i in range(len(random_seeds)):
    np.random.seed(random_seeds[i])
    print_value_trajectory(root_states[i])
    print('------------------------------------------------------------------')
```

```
 x | _ | _
-----------
 _ | _ | _
-----------
 _ | _ | _


Value of the above state is 9.5626
 x | _ | _
-----------
 _ | _ | o
-----------
 _ | x | _
```

```
Value of the above state is 9.514000000000001
 x | o | _
-----------
 _ | _ | o
-----------
 x | x | _


Value of the above state is -1.4
 x | o | _
-----------
 x | _ | o
-----------
 x | x | o


Value of the above state is -10.0
------------------------------------------------------------------
 _ | x | _
-----------
 _ | _ | _
-----------
 _ | _ | _


Value of the above state is 9.736171428571428
 _ | x | o
-----------
 _ | _ | _
-----------
 _ | x | _


Value of the above state is 10.0
 x | x | o
-----------
 _ | _ | _
-----------
 o | x | _


Value of the above state is 10.0
 x | x | o
-----------
 _ | o | _
-----------
 o | x | _


Value of the above state is 10.0
------------------------------------------------------------------
 _ | _ | x
-----------
 _ | _ | _
-----------
 _ | _ | _


Value of the above state is 9.5626
 _ | _ | x
-----------
 x | o | _
-----------
 _ | _ | _


Value of the above state is 9.514000000000001
 _ | _ | x
-----------
```

```
 x | o | x
-----------
 _ | o | _


Value of the above state is 10.0
 _ | o | x
-----------
 x | o | x
-----------
 _ | o | _


Value of the above state is 10.0
------------------------------------------------------------------
 _ | _ | _
-----------
 x | _ | _
-----------
 _ | _ | _


Value of the above state is 9.736171428571428
 _ | _ | _
-----------
 x | _ | _
-----------
 o | x | _


Value of the above state is 9.514000000000001
 _ | _ | _
-----------
 x | _ | x
-----------
 o | x | o


Value of the above state is 10.0
 o | _ | _
-----------
 x | x | x
-----------
 o | x | o


Value of the above state is -10.0
------------------------------------------------------------------
 _ | _ | _
-----------
 _ | x | _
-----------
 _ | _ | _


Value of the above state is 8.312885714285715
 _ | _ | _
-----------
 x | x | _
-----------
 _ | o | _


Value of the above state is 5.41
 x | _ | _
-----------
 x | x | _
-----------
 _ | o | o
```

```
Value of the above state is 10.0
 x | o | _
-----------
 x | x | _
-----------
 x | o | o


Value of the above state is -10.0
------------------------------------------------------------------
```

The arbitrary initial values chosen for value iteration for all the states are 0. As value iteration follows contraction mapping, as the iterations -> ∞ (infinity), the difference between the true value of the state and the current value -> 0 and they converge at a point (which is the true value of the state). Since we have taken a convergence criteria of , $\|V - V*\|_\infty \leq 0.1$, the updates stop after this point (once the difference hits <= 0.1). Hence the values of the states after the value iteration differ no more than 0.1 from the true values.

Question 4

In [10]:

```
def print_Q_states(parent):

    index = states_keys.index(parent)
    #print('Q Value of the above state is ' + str(rewards_list[index] + gamma * values[index]))

    available_moves = valid_moves(parent)

    for move in available_moves:
        s = list(parent)
        s[move] = 'o'
        s = ''.join(s)

        reward = check_win(s)
        if reward != 1:
            print('Action for O is at position ' + str(move + 1) + ' and the Q value is ' + str(values[states_keys.index(s)]))
            continue

        total = 0

        for position in valid_moves(s):

            new_parent = list(s)
            new_parent[position] = 'x'
            new_parent = ''.join(new_parent)

            r = check_win(new_parent)


            total += (probability_table[move, index, states_keys.index(new_parent)] * values[states_keys.index(new_parent)])

        total *= gamma
        total += rewards_list[states_keys.index(parent)]

        print('Action for O is at position ' + str(move + 1) + ' and the Q value is ' + str(total))



for i in range(3,5):
    print('Current state ' + str(list(root_states[i])))
    print_Q_states(root_states[i])
    print('------------------------------------------------------------------')
```

```
Current state ['_', '_', '_', 'x', '_', '_', '_', '_', '_']
Action for O is at position 1 and the Q value is 9.187685714285713
```

```
Action for O is at position 2 and the Q value is 8.896085714285714
Action for O is at position 3 and the Q value is 9.736171428571428
Action for O is at position 5 and the Q value is 9.625085714285715
Action for O is at position 6 and the Q value is 8.285114285714286
Action for O is at position 7 and the Q value is 9.187685714285713
Action for O is at position 8 and the Q value is 8.896085714285714
Action for O is at position 9 and the Q value is 9.736171428571428
------------------------------------------------------------
Current state ['_', '_', '_', '_', 'x', '_', '_', '_', '_']
Action for O is at position 1 and the Q value is 8.312885714285715
Action for O is at position 2 and the Q value is 6.924314285714286
Action for O is at position 3 and the Q value is 8.312885714285713
Action for O is at position 4 and the Q value is 6.924314285714285
Action for O is at position 6 and the Q value is 6.924314285714286
Action for O is at position 7 and the Q value is 8.312885714285713
Action for O is at position 8 and the Q value is 6.924314285714286
Action for O is at position 9 and the Q value is 8.312885714285713
------------------------------------------------------------
```

In [ ]:

Question 5

Training agent for c = 2 i.e. c > 0

In [1]:
```
import numpy as np
import copy
from collections import defaultdict

WINNING_POSITIONS = [(0,1,2), (3,4,5), (6,7,8), (0,3,6), (1,4,7), (2,5,8), (0,4,8),(2,4,6)]
gamma = 0.9
X_PLAYER, O_PLAYER = 1, 2

def check_win(state):
    for a, b, c in WINNING_POSITIONS:
        # player - 1- 'X' wins, R(s) = -10
        if state[a] == state[b] == state[c] and state[a] == 'x':
            return -10
        # player - 2- 'O' wins, R(s) = 10
        elif state[a] == state[b] == state[c] and state[a] == 'o':
            return 10

    # draw
    if len(valid_moves(state)) == 0:
        return 0
    # non terminal state
    else:
        return 2

def valid_moves(state):
    s = np.array(list(state))
    return np.asarray(np.where(s == '_')).flatten()


initial_state = '_________'
states = defaultdict()
root_states = []


def generate_child_states(parent):

    available_moves = valid_moves(parent)

    for move in available_moves:
        s = list(parent)
        s[move] = 'o'
        s = ''.join(s)

        reward = check_win(s)
        if reward != 2:
            states[s] = reward
            continue

        for position in valid_moves(s):
            new_parent = list(s)
            new_parent[position] = 'x'
            new_parent = ''.join(new_parent)

            r = check_win(new_parent)

            states[new_parent] = r

            if r != 2:
                continue
            else:
```

```
                generate_child_states(new_parent)
    return


# Generate roots
def generate_initial_states():

    for i in range(9):
        s = list(initial_state)
        s[i] = 'x'
        s = ''.join(s)
        root_states.append(s)

        states[s] = check_win(s)

        # Generate child states
        generate_child_states(s)


generate_initial_states()
```

In [2]:
```
states_keys = list(states.keys())
probability_table = np.zeros((9, len(states), len(states)))
rewards_list = np.zeros(len(states))
```

In [3]:
```
def transition_probabilities(parent):
    available_moves = valid_moves(parent)
    parent_index = states_keys.index(parent)

    for move in available_moves:
        s = list(parent)
        s[move] = 'o'
        s = ''.join(s)

        if check_win(s) != 2:
            rewards_list[states_keys.index(s)] = states[s]
            probability_table[move, parent_index, states_keys.index(s)] = 1
            continue

        for position in valid_moves(s):
            new_parent = list(s)
            new_parent[position] = 'x'
            new_parent = ''.join(new_parent)

            probability_table[move, parent_index, states_keys.index(new_parent)] = 1/len(valid_moves(s))
            rewards_list[states_keys.index(new_parent)] = states[new_parent]


            if check_win(new_parent) != 2:
                continue
            else:
                transition_probabilities(new_parent)
    return
```

In [4]:
```
for root in root_states:
    rewards_list[states_keys.index(root)] = states[root]
    transition_probabilities(root)
```

In [6]:
```
values = np.zeros(len(states))
actions = [0 for i in range(len(states))]


def check_for_convergence(old_vals):
    diff = values - old_vals

    if np.linalg.norm(abs(diff), np.inf) < 0.1:
        return True;
    else:
        return False;


while True:
    old_vals = copy.deepcopy(values)

    for i in range(len(states)):
        win = check_win(states_keys[i])
        if win != 2:
            values[i] = rewards_list[i]
            continue

        maximum = -1000
        action = 0

        for j in range(9):

            if np.sum(probability_table[j, i,:]) == 0:
                continue

            new_val = rewards_list[i] + gamma * (np.dot(probability_table[j, i, :], values))

            if new_val > maximum:
                maximum = new_val
                action = j

        values[i] = maximum
        actions[i] = action

    if check_for_convergence(old_vals) == False:
        continue;
    else:
        break;
```

In [7]:
```
print(set(values))
```

```
{0.20000000000000018, 0.0, 2.0, 3.8, 0.19999999999999996, 5.6, 2.9, -0.6999999999999997, 6.5, 9.2, 10.0, 11.9, 12.730828571428571, 11.0, 10.604000000000001, 7.04, 7.526, 9.146, 12.062000000000001, 12.548000000000002, 11.252, 12.062, 1.1, 8.66, 8.822, 8.012, 10.766000000000002, 10.766, 10.604000000000003, 11.900000000000002, 11.576, 11.576000000000002, 11.481114285714286, 12.548, 12.95994285714286, 12.710000000000003, 12.959942857142858, 0.2000000000000004, 7.526000000000001, -10.0, -7.0, -6.1}
```

In [8]:
```
def print_Q_states(parent):

    index = states_keys.index(parent)
    #print('Q Value of the above state is ' + str(rewards_list[index] + gamma * values[index]))

    available_moves = valid_moves(parent)

    for move in available_moves:
        s = list(parent)
        s[move] = 'o'
        s = ''.join(s)

        reward = check_win(s)
        if reward != 2:
            print('Action for O is at position ' + str(move + 1) + ' and the Q value is ' + str(values[states_keys.index(s)]))
            continue

        total = 0

        for position in valid_moves(s):

            new_parent = list(s)
            new_parent[position] = 'x'
            new_parent = ''.join(new_parent)

            r = check_win(new_parent)


            total += (probability_table[move, index, states_keys.index(new_parent)] * values[states_keys.index(new_parent)])

        total *= gamma
        total += rewards_list[states_keys.index(parent)]

        print('Action for O is at position ' + str(move + 1) + ' and the Q value is ' + str(total))

print('Generating Q states for state x_xoo_x__ with c = 2\n')
print_Q_states('x_xoo_x__')
```

```
Generating Q states for state x_xoo_x__ with c = 2

Action for O is at position 2 and the Q value is 11.9
Action for O is at position 6 and the Q value is 10.0
Action for O is at position 8 and the Q value is 5.6
Action for O is at position 9 and the Q value is 2.9
```

In [ ]:
```

```

Question 5

Training agent for c = -1 i.e. c < 0

In [1]:
```
import numpy as np
import copy
from collections import defaultdict

WINNING_POSITIONS = [(0,1,2), (3,4,5), (6,7,8), (0,3,6), (1,4,7), (2,5,8), (0,4,8),(2,4,6)]
gamma = 0.9
X_PLAYER, O_PLAYER = 1, 2

def check_win(state):
    for a, b, c in WINNING_POSITIONS:
        # player - 1- 'X' wins, R(s) = -10
        if state[a] == state[b] == state[c] and state[a] == 'x':
            return -10
        # player - 2- 'O' wins, R(s) = 10
        elif state[a] == state[b] == state[c] and state[a] == 'o':
            return 10

    # draw
    if len(valid_moves(state)) == 0:
        return 0
    # non terminal state
    else:
        return -1

def valid_moves(state):
    s = np.array(list(state))
    return np.asarray(np.where(s == '_')).flatten()


initial_state = '_________'
states = defaultdict()
root_states = []


def generate_child_states(parent):

    available_moves = valid_moves(parent)

    for move in available_moves:
        s = list(parent)
        s[move] = 'o'
        s = ''.join(s)

        reward = check_win(s)
        if reward != -1:
            states[s] = reward
            continue

        for position in valid_moves(s):
            new_parent = list(s)
            new_parent[position] = 'x'
            new_parent = ''.join(new_parent)

            r = check_win(new_parent)

            states[new_parent] = r

            if r != -1:
                continue
            else:
```

```
                generate_child_states(new_parent)
    return


# Generate roots
def generate_initial_states():

    for i in range(9):
        s = list(initial_state)
        s[i] = 'x'
        s = ''.join(s)
        root_states.append(s)

        states[s] = check_win(s)

        # Generate child states
        generate_child_states(s)


generate_initial_states()
```

In [2]:
```
states_keys = list(states.keys())
probability_table = np.zeros((9, len(states), len(states)))
rewards_list = np.zeros(len(states))
```

In [3]:
```
def transition_probabilities(parent):
    available_moves = valid_moves(parent)
    parent_index = states_keys.index(parent)

    for move in available_moves:
        s = list(parent)
        s[move] = 'o'
        s = ''.join(s)

        if check_win(s) != -1:
            rewards_list[states_keys.index(s)] = states[s]
            probability_table[move, parent_index, states_keys.index(s)] = 1 # double check this
            continue

        for position in valid_moves(s):
            new_parent = list(s)
            new_parent[position] = 'x'
            new_parent = ''.join(new_parent)

            probability_table[move, parent_index, states_keys.index(new_parent)] = 1/len(valid_moves(s)) # double check this
            rewards_list[states_keys.index(new_parent)] = states[new_parent]


            if check_win(new_parent) != -1:
                continue
            else:
                transition_probabilities(new_parent)
    return
```

In [4]:
```
for root in root_states:
    rewards_list[states_keys.index(root)] = states[root]
    transition_probabilities(root)
```

In [5]:
```
values = np.zeros(len(states))
actions = [0 for i in range(len(states))]


def check_for_convergence(old_vals):
    diff = values - old_vals

    if np.linalg.norm(abs(diff), np.inf) < 0.1:
        return True
    else:
        return False


while True:
    old_vals = copy.deepcopy(values)

    for i in range(len(states)):
        win = check_win(states_keys[i])
        if win != -1:
            values[i] = rewards_list[i]
            continue

        maximum = -1000
        action = 0

        for j in range(9):

            if np.sum(probability_table[j, i,:]) == 0:
                continue

            new_val = rewards_list[i] + gamma * (np.dot(probability_table[j, i, :], values))

            if new_val > maximum:
                maximum = new_val
                action = j

        values[i] = maximum
        actions[i] = action

    if check_for_convergence(old_vals) == False:
        continue;
    else:
        break;
```

In [6]:
```
print(set(values))
```

```
{0.0, 0.8, 2.636000000000001, 3.851000000000001, 3.5, 5.3900000000000015, 6.2, -0.7659999999999999, 8.0, 5.39, 10.0, 5.876, 4.904000000000001, 4.418, 5.390000000000001, 2.3930000000000007, 0.6920000000000004, 4.418000000000001, 4.038457142857142, 1.1780000000000008, 0.6920000000000008, -0.766, 1.178, 4.038457142857143, 1.1780000000000004, -10.0, -4.6, -1.0, -1.9}
```

In [7]:
```
def print_Q_states(parent):

    index = states_keys.index(parent)
    #print('Q Value of the above state is ' + str(rewards_list[index] + gamma * values[index]))

    available_moves = valid_moves(parent)

    for move in available_moves:
        s = list(parent)
        s[move] = 'o'
        s = ''.join(s)

        reward = check_win(s)
        if reward != -1:
            print('Action for O is at position ' + str(move + 1) + ' and the Q value is ' + str(values[states_keys.index(s)]))
            continue

        total = 0

        for position in valid_moves(s):

            new_parent = list(s)
            new_parent[position] = 'x'
            new_parent = ''.join(new_parent)

            r = check_win(new_parent)


            total += (probability_table[move, index, states_keys.index(new_parent)] * values[states_keys.index(new_parent)])

        total *= gamma
        total += rewards_list[states_keys.index(parent)]

        print('Action for O is at position ' + str(move + 1) + ' and the Q value is ' + str(total))



print('Generating Q states for state x_xoo_x__ with c = -1\n')
print_Q_states('x_xoo_x__')
```

```
Generating Q states for state x_xoo_x__ with c = -1

Action for O is at position 2 and the Q value is 6.2
Action for O is at position 6 and the Q value is 10.0
Action for O is at position 8 and the Q value is 0.8
Action for O is at position 9 and the Q value is -1.9
```

In [ ]:

2 Function approximation in RL

Q7

(1) Assume $R(s_i) = 10 \quad \forall s$

We know

$$Q(s,a) = R(s) + \gamma \sum_{s'} P(s'|s,a) V^*(s')$$

Since s_1, s_2, s_3 are terminal states, $V^*(s_1) = R(s_1)$,

$V^*(s_2) = R(s_2)$ and $V^*(s_3) = R(s_3)$

$$\Rightarrow Q(s_0, a_1) = R(s_0) + \gamma \left[P(s_1|s_0,a_1) V^*(s_1) + P(s_2|s_0,a_1) V^*(s_2) \right]$$

$$= 10 + 0.5 \left(0.5 \times 10 + 0.5 \times 10 \right) = 15$$

$$Q(s_0, a_2) = R(s_0) + \gamma \left(P(s_2|s_0,a_2) V^*(s_2) \right)$$

$$= 10 + 0.5 (1 \times 10) = 15$$

$$Q(s_0, a_3) = R(s_0) + \gamma \left(P(s_3|s_0,a_3) V^*(s_3) \right)$$

$$= 10 + (0.5 \times 1 \times 10) = 15$$

~~Assume~~ By the linear Q fn, $Q(s,a) = \alpha s + \beta a$

$$Q(s_0, a_1) = \alpha(1) + \beta(1) = \alpha + \beta$$

$$Q(s_0, a_2) = \alpha(1) + \beta(2) = \alpha + 2\beta$$

$$Q(s_0, a_3) = \alpha(1) + \beta(3) = \alpha + 3\beta$$

$\Rightarrow \alpha + \beta = \alpha + 2\beta = \alpha + 3\beta = 15$ (substituting from above)

$\Rightarrow \alpha = 15 \quad \beta = 0$

Hence for $\alpha = 1.5$ $\beta = 0$ and $R(s) = 10 \ \forall s$, the linear Q function correctly represents all values at s_0

(2) Assume $R(s_i) = 2i$

$\Rightarrow R(s_0) = 0 \quad R(s_1) = 2 \quad R(s_2) = 4 \quad R(s_3) = 6$

$\Rightarrow V^*(s_1) = 2 \quad V^*(s_2) = 4 \quad V^*(s_3) = 6$

Now

$$Q(s_0, a_1) = 0 + 0.5\left(0.5 \times 2 + 0.5 \times 4\right)$$

$$= 1.5$$

$$Q(s_0, a_2) = 0 + 0.5(1 \times 4) = 2$$

$$Q(s_0, a_3) = 0 + 0.5 \times 1 \times 6 = 3$$

Now we know from (1) that $\alpha + \beta = 1.5$

$\alpha + 2\beta = 2 \qquad \alpha + 3\beta = 3$

Solving $\alpha + \beta = 1.5$ and $\alpha + 2\beta = 2$ by elimination method, we get, $\beta = 0.5 \quad \alpha = 1$

But substituting this in $\alpha + 3\beta = 3$ gives,

$1 + 3 \times 0.5 = 2.5 \neq 3$

$\Rightarrow$ this is a non-solvable set of equations without a unique solution. The difference ~~in~~ between the ~~Reward~~ function based values and the true Q value will increase with the increase in state space and no value of α and β can satisfy the true Q values.

(3) We know $Q(s_0, a_1) = 1.5$, $Q(s_0, a_2) = 2$,

$Q(s_0, a_3) = 3$

Assume the function :- $Q(s,a) = \alpha a(a-1) + \beta s$

~~With s=1~~ : $\alpha = 0.25$ $\beta = 1.5$ (calculated using online solver)

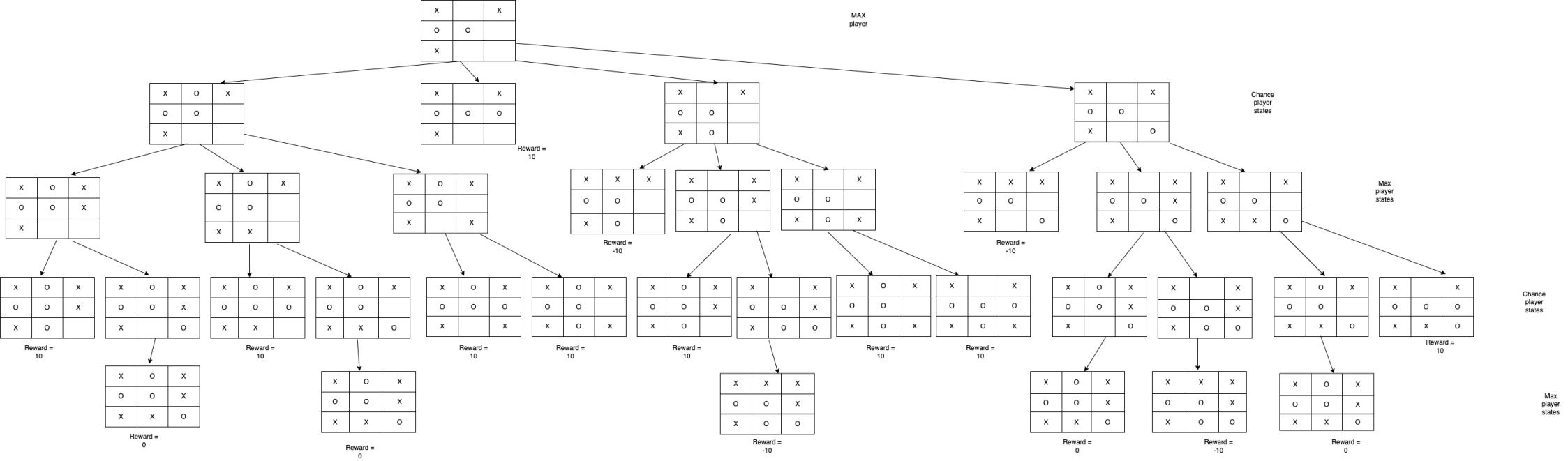
Now,

$Q(s_0, a_1) = 0.25 \times 1 (1-1) + 1.5 \times 1 = 1.5$

$Q(s_0, a_2) = 0.25 \times 2 (2-1) \times 1.5 \times 1 = 2$

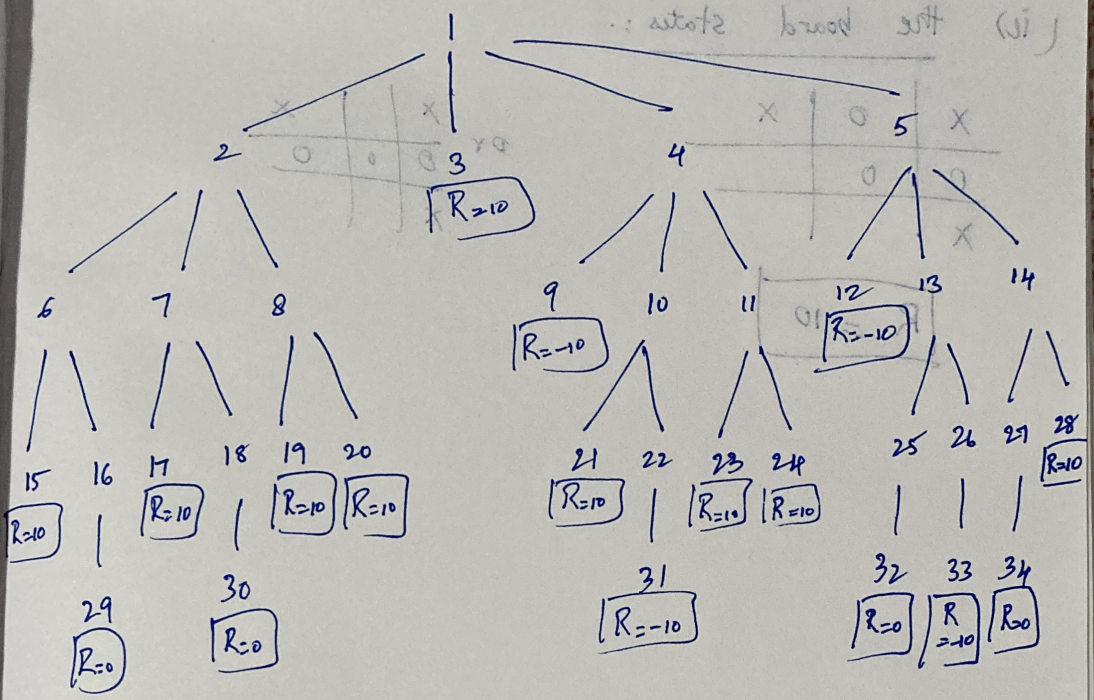
$Q(s_0, a_3) = 0.25 \times 3 (3-1) \times 1.5 \times 1 = 3$

which matches with the true Q values calculated with the reward function $R(s_i) = 2i$


MAX player
Chance player states
Max player states
Chance player states
Max player states
Reward = 10
Reward = -10
Reward = 0

Question 6

Numbering the states for easier explanation



Value at 16 = 0 (since 29 is the only child state)

Value at 18 = 0 (since 30 is the only child state)

Value at 6 = max(10, 0) = 10

Value at 7 = max(10, 0) = 10

Value at 8 = max(10, 10) = 10

Value at 2 = $\frac{10}{3} + \frac{10}{3} + \frac{10}{3} = 10$ (chance player)

Value at 3 = 10

Since the subtrees of 4 and 5 contain 0 and -10, their max values will be < 10.

⇒ max value chosen by 1 is 10

Optimal action

either 2 or 3

(ii) the board states :-