Adversarial Search: Playing Connect 4

Student Name: [Cagatay Duygu - 48369962]

Instructions

Total Points: Undegraduates 100, graduate students 110

Complete this notebook and submit it. The notebook needs to be a complete project report with your implementation, documentation including a short discussion of how your implementation works and your design choices, and experimental results (e.g., tables and charts with simulation results) with a short discussion of what they mean. Use the provided notebook cells and insert additional code and markdown cells as needed.

Introduction

You will implement different versions of agents that play Connect 4:

> "Connect 4 is a two-player connection board game, in which the players choose a color and then take turns dropping colored discs into a seven-column, six-row vertically suspended grid. The pieces fall straight down, occupying the lowest available space within the column. The objective of the game is to be the first to form a horizontal, vertical, or diagonal line of four of one's own discs." (see Connect Four on Wikipedia)

Note that Connect-4 has been solved in 1988. A connect-4 solver with a discussion of how to solve different parts of the problem can be found here: https://connect4.gamesolver.org/en/

Task 1: Defining the Search Problem [10 point]

Define the components of the search problem:

- Initial state
- Actions
- Transition model (result function)
- Goal state (terminal state and utility)

Your code/answer goes here.

Initial state is the state the agent starts working. State can be defined with two parameter in our case: positions of disks and board itself. The initial state will be expressed as empty board (6x7 board, all positions are 0.).

Actions: All options that can be taken in the given state. Actions are the columns that has at least one empty row for Connect 4 game.

Transition model is a result of what each action does. Transition models is a function that has the input of state and action and has the ouput of next state.

The Connect Four environment is not deterministic, there is an opponent. The transition model will give the result state from the action taken by the agent and the current state in their turn. When the other turn of the agent comes, it is not the same with the result from transition model last turn. It is the state after the opponent plays.

Goal State is the desired final state. The goal state is having 4 disks from agent's color in a row either vertically, horizontally or diagonally. Both a loss or a draw is not the goal state. Thus, it can be considered as fail if it rusult is seen binary.

How big is the state space? Give an estimate and explain it.

We can do a approximation for the state space. Since we have 6x7=42 positions and 3 possible positions, we can guess the state-space size as 3^42.

However, this value ignores the gravity effect and has lots of illegal states.

The real state space value is much smaller than this value.

The best value for the size of state-space of Connect-4 has been calculated by a computer program is around 1.6 * 10^13. [1]

[1] MIT. “Connect 4.” Connect 4, MIT, web.mit.edu/sp.268/www/2010/connectFourSlides.pdf.

How big is the game tree that minimax search will go through? Give an estimate and explain it.

Maximum Depth of Tree: m=42 (6x7)

Branching Factor: b=7 (Column Size)

Time: $O(bm)$ = 7^42

Space: $O(bm)$ = 7*42

Task 2: Game Environment and Random Agent [25 point]

Use a numpy character array as the board.

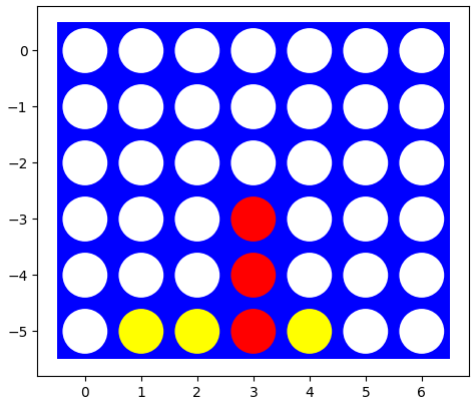
In [170…
```
import numpy as np
import pandas as pd
import math

def empty_board(shape=(6, 7)):
    return np.full(shape=shape, fill_value=0)

print(empty_board())
```

```
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]]
```

The standard board is 6×7 but you can use smaller boards to test your code. Instead of colors (red and yellow), I use 1 and -1 to represent the players. Make sure that your agent functions all have the from: `agent_type(board, player = 1)`, where board is the current board position (in the format above) and player is the player whose next move it is and who the agent should play (as 1 and -1).

In [196…
```
# Visualization code by Randolph Rankin

import matplotlib.pyplot as plt

def visualize(board):
```

```
    plt.axes()
    rectangle=plt.Rectangle((-0.5,len(board)*-1+0.5),len(board[0]),len(board),fc='blue')
    circles=[]
    for i,row in enumerate(board):
        for j,val in enumerate(row):
            color='white' if val==0 else 'red' if val==1 else 'yellow'
            circles.append(plt.Circle((j,i*-1),0.4,fc=color))

    plt.gca().add_patch(rectangle)
    for circle in circles:
        plt.gca().add_patch(circle)

    plt.axis('scaled')
    plt.show()

board = [[0, 0, 0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 0, 0],
         [0, 0, 0, 1, 0, 0, 0],
         [0, 0, 0, 1, 0, 0, 0],
         [0,-1,-1, 1,-1, 0, 0]]
visualize(board)
```



Implement helper functions for:

- A check for available actions in each state `actions(s)`.
- The transition model `result(s, a)`.
- Check for terminal states `terminal(s)`.
- The utility function `utility(s)`.

Make sure that all these functions work with boards of different sizes (number of columns and rows).

Helper Functions:

I used the tic-tac-toe example code and change it accordingly for 4-Connect case.

In [230…
```
# possible moves
def actions(state):
    actions = list()

    for i in np.where(np.array(state[0]) == 0)[0]:
        actions.append(i)

    return actions
```

In [227…
```
def empty_row_lowest(state, column):
    no_of_rows = len(state)
    for j in range(no_of_rows):
        if state[no_of_rows - j - 1][column] == 0:

            #print("no_of_rows:", no_of_rows, "column:", column )

            empty_row = no_of_rows - j - 1
            return empty_row
```

In [197…
```
empty_row_lowest(board, 3)
```

Out[197]: 2

In [175…
```
def switch(player):
    if player==1: return -1
    else: return 1
```

In [235…
```
#transitional model
def action_result(state, action, player):
    state = state.copy()

    #print(state)

    pos = empty_row_lowest(state, action)

    #print("pos:", pos), print("action:",action)

    state[pos][action] = player
    return state

def result(state, action, player):
    state = state.copy()

    #move
    state = action_result(state, action, player)

    # check the new available actions (for the opponent)
    result=[]
    available_actions_opponent = actions(state)
```

```
    #check if the state is full or not
    if len(available_actions_opponent) < 1 :
        return [state]
    for opponent_action in available_actions_opponent:
        possible_state = action_result(state, opponent_action, switch(player))
        result.append(possible_state)
    return result
```

In [225…

```
""" This function returns utility and checks the terminal I did not write
2 different functions for these 2. """

def terminal(state):
    state=state.copy()
    w = len(state[0]) #width of the state
    h = len(state) #height of the state

    # horizontal terminal
    for i in range(h):
        for j in range(w - 3):
            if(state[i][j] + state[i][j + 1] + state[i][j + 2] + state[i][j + 3] == 4):
                return 1
            if(state[i][j] + state[i][j + 1] + state[i][j + 2] + state[i][j + 3] == -4):
                return -1

    # vertical terminal
    for j in range(w):
        for i in range(h - 3):
            if(state[i][j] + state[i + 1][j] + state[i + 2][j] + state[i + 3][j] == 4):
                return 1
            if(state[i][j] + state[i + 1][j] + state[i + 2][j] + state[i + 3][j] == -4):
                return -1

    # diagonal terminal
    for i in range(h - 3):
        for j in range(w - 3):

            if(state[i][j] + state[i + 1][j + 1] + state[i + 2][j + 2] + state[i + 3][j + 3] == -4):
                return -1

            if(state[i + 3][j] + state[i + 2][j + 1] + state[i + 1][j + 2] + state[i][j + 3] == -4):
                return -1

            if(state[i][j] + state[i + 1][j + 1] + state[i + 2][j + 2] + state[i + 3][j + 3] == 4):
                return 1

            if(state[i + 3][j] + state[i + 2][j + 1] + state[i + 1][j + 2] + state[i][j + 3] == 4):
                return 1

    # check if the state is full (draw)
    if 0 not in state:
        return 0
    return None
```

Implement an agent that plays randomly. Make sure the agent function receives as the percept the board and returns a valid action. Use an agent function definition with the following signature (arguments):

`def random_player(board, player = 1): ...`

The argument `player` is used for agents that do not store what color they are playing. The value passed on by the environment should be 1 ot -1 for player red and yellow, respectively. See Experiments section for tic-tac-toe for an example.

In [238…

```
def random_player(state):
    action = actions(state)
    random_choice = np.random.randint(0, len(action))
    return action[random_choice]
```

In [179…

```
# I defined a dummy alpha_beta_minimax
#value since I use it in the game function
#I wanted to generalize the game function
alpha_beta_minimax=0
```

In [232…

```
def switch_player(player, algorithm1, algorithm2):
    if player == 1:
        return -1, algorithm2
    else:
        return 1, algorithm1

def game(player1, player2, first_player, Verbose, n = 1000 , h=6, w=7):
    result_record = {1: 0, -1: 0, 0: 0}
    #n plays between 2 players
    for i in range(n):
        iters = 0
        state = empty_board(shape=(h, w))
        w=len(state[0])
        h=len(state)
        player=first_player
        current_player = player1

        # num of iteration during 1 game
        while iters < w*h+1:
            iters += 1

            if(current_player == alpha_beta_minimax):
                action = current_player(state, 10, player)
                # 10 is the cutoff depth, will be defined in the minimax part.
            else:
                action = current_player(state)

            state = action_result(state, action, player)
            if Verbose==True:
                visualize(state)
            winner = terminal(state)

            if winner != None:
                result_record[winner] += 1
                break

            player, current_player = switch_player(player, player1, player2)

    return result_record
```

Random vs Random

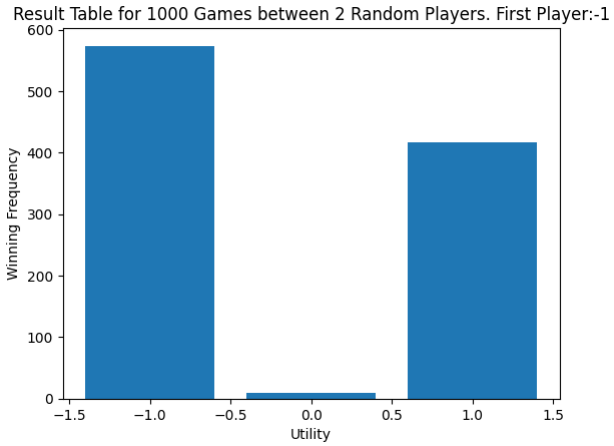
Let two random agents play against each other 1000 times. Look at the Experiments section for tic-tac-toe to see how the environment uses the agent functions to play against each other.

How often does each player win? Is the result expected?

In [18]:
```
first_player=-1#First Player -1
result_random_vs_random=pd.DataFrame([game(random_player, random_player,first_player,False)])
print(result_random_vs_random)

result_arr = result_random_vs_random.to_numpy()
utility_arr=([1, -1, 0])
plt.bar(utility_arr,result_arr[0])
plt.xlabel("Utility")
plt.ylabel("Winning Frequency")
plt.title("Result Table for 1000 Games between 2 Random Players. First Player:"+str(first_player))
```

```
   1   -1   0
0  417  574   9
```

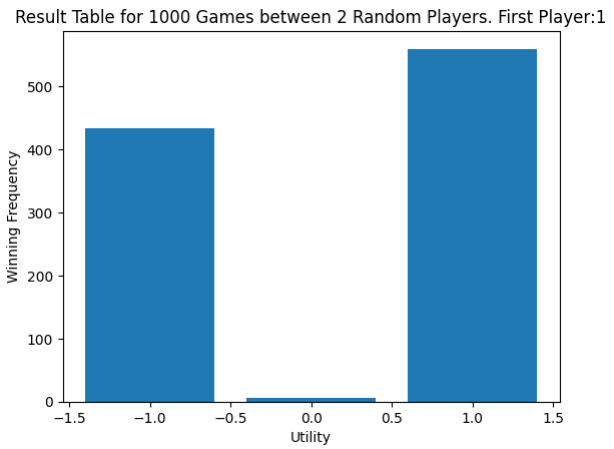
Out[18]: Text(0.5, 1.0, 'Result Table for 1000 Games between 2 Random Players. First Player:-1')



In [19]:
```
first_player=1#First Player 1
result_random_vs_random=pd.DataFrame([game(random_player, random_player,first_player, False)])
print(result_random_vs_random)

result_arr = result_random_vs_random.to_numpy()
utility_arr=([1, -1, 0])
plt.bar(utility_arr,result_arr[0])
plt.xlabel("Utility")
plt.ylabel("Winning Frequency")
plt.title("Result Table for 1000 Games between 2 Random Players. First Player:"+str(first_player))
```

```
   1   -1   0
0  560  434   6
```

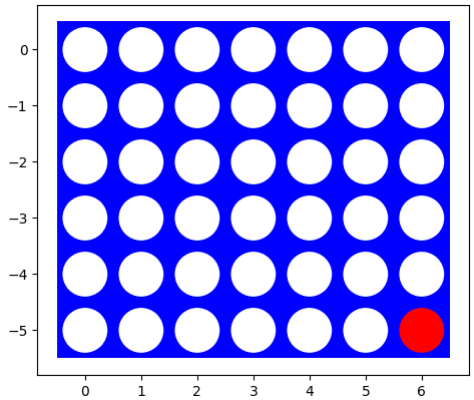
Out[19]: Text(0.5, 1.0, 'Result Table for 1000 Games between 2 Random Players. First Player:1')

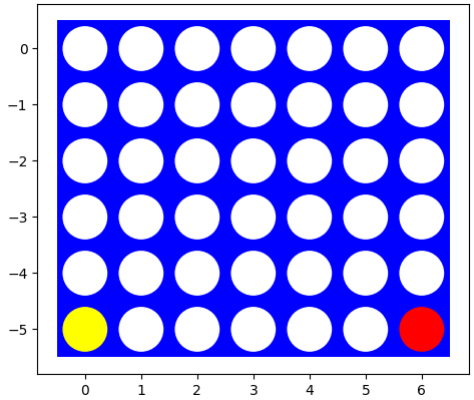


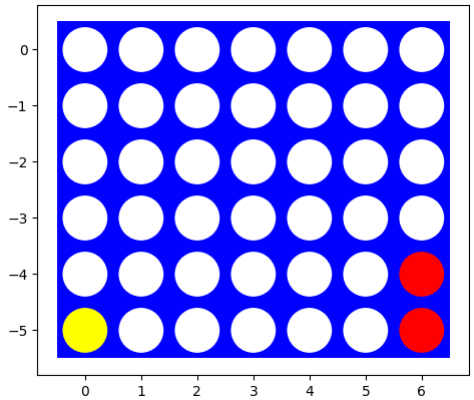
Agent that moves first has an advantage over the second one (approximately 57%). This can be seen easily from the graphs. This behavior is expected since if the two players play optimally, the first one always wins.

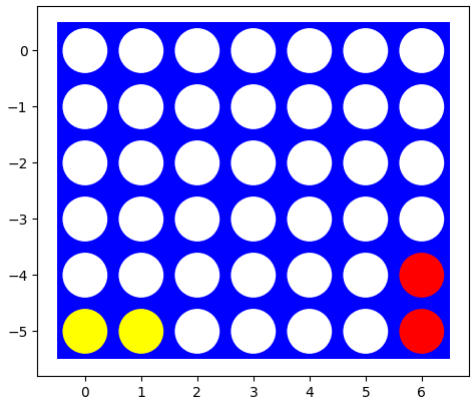
Example Game, Random vs Random

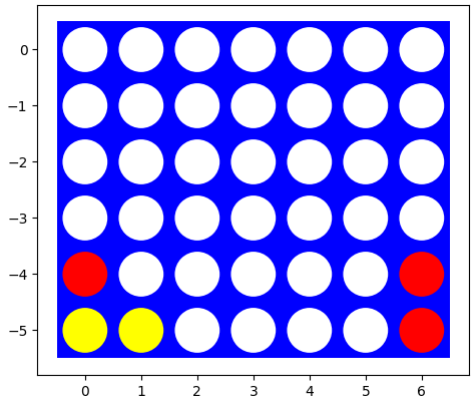
In [20]:
```
game(random_player, random_player,1, True, n=1)
```

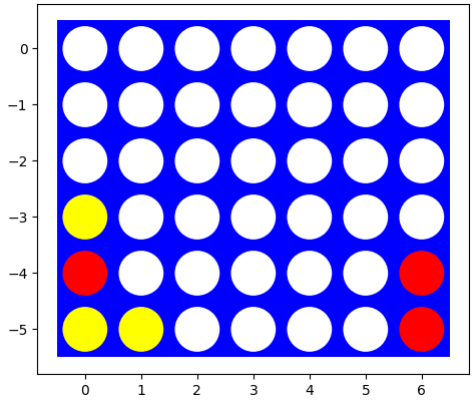

0
−1
−2
−3
−4
−5
0 1 2 3 4 5 6

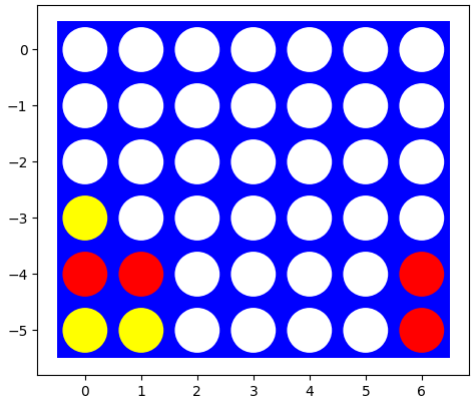

0
−1
−2
−3
−4
−5
0 1 2 3 4 5 6

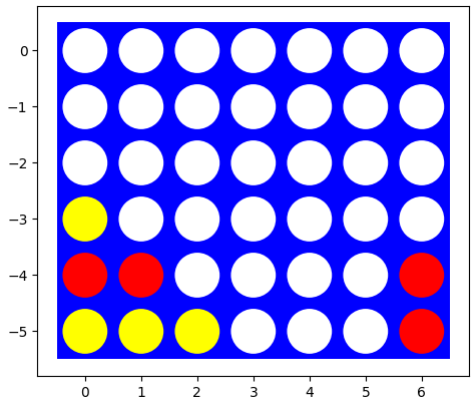

0
−1
−2
−3
−4
−5
0 1 2 3 4 5 6

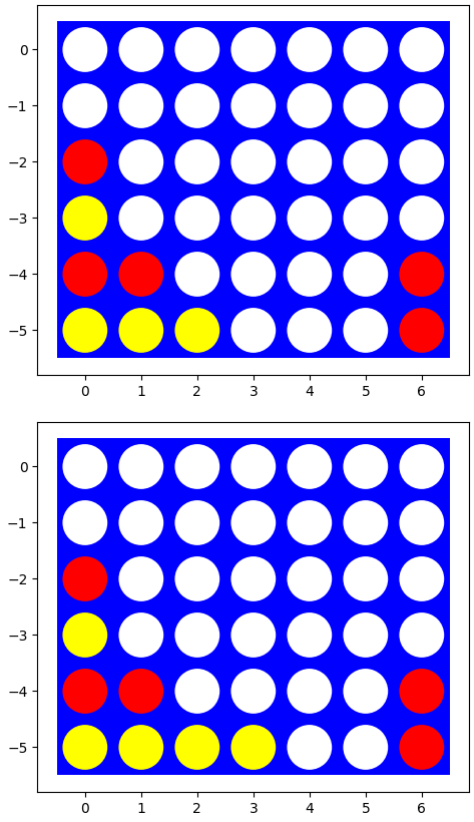

0
−1
−2
−3
−4
−5
0 1 2 3 4 5 6


0
−1
−2
−3
−4
−5
0 1 2 3 4 5 6


0
−1
−2
−3
−4
−5
0 1 2 3 4 5 6


0
−1
−2
−3
−4
−5
0 1 2 3 4 5 6


0
−1
−2
−3
−4
−5
0 1 2 3 4 5 6



Out[20]: {1: 0, -1: 1, 0: 0}

Task 3: Minimax Search with Alpha-Beta Pruning

Implement the Search [20 points]

Implement minimax search starting from a given board for specifying the player. You can use code from the tic-tac-toe example.

Important Notes:

- Make sure that all your agent functions have a signature consistent with the random agent above and that it uses a class to store state information.

This is essential to be able play against agents from other students later.

- The search space for a 6×7 board is large. You can experiment with smaller boards (the smallest is 4×4) and/or changing the winning rule to connect 3 instead of 4.

In [229…

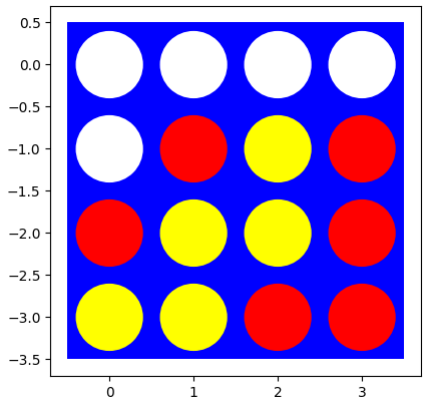
```
def max_value_alpha_beta(limit_moves, deep, state, player_id, alpha, beta):
    deep += 1

    # terminal state check
    val = terminal(state)
    if val is not None:
        return val, None
    if deep > limit_moves:
        return 0, None

    val, move = -math.inf, None
    # check all the moves in the state,
    # change the alpha, return largest utility move
    for action in actions(state):
        val2, act2 = min_value_alpha_beta(limit_moves, deep, action_result(state, action, player_id), player_id, alpha, beta)
        if val2 > val:
            val, move = val2, action
            alpha = max(alpha, val)
        if val >= beta: return val, move

    return val, move

def min_value_alpha_beta(limit_moves, deep, state, player_id, alpha, beta):
    deep += 1

    # check if the limit is achieved or
    # it is the terminal state
    val = terminal(state)
    if val is not None:
        return val, None
    # i added a cutoff here
    if deep > limit_moves:
        return 0, None

    val, move = +math.inf, None
    # check all the moves in the state,
    # change the beta, return smallest utility move
    val2, act2 = max_value_alpha_beta(limit_moves, deep, action_result(state, action, switch(player_id)), player_id, alpha, beta)
    if val2 < val:
        val, move = val2, action
        beta = min(beta, val)
    if val <= alpha:
        return val, move

    return val, move
```

```
def alpha_beta_minimax(state, limit_moves = +math.inf, player_id = 1):
    deep = 0
    val, move = max_value_alpha_beta(limit_moves, deep, state, player_id, -math.inf, +math.inf)
    return move
```

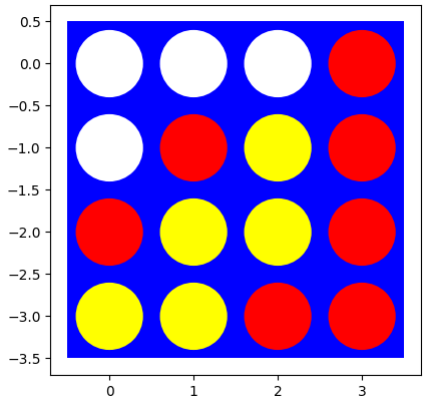
Experiment with some manually created boards (at least 5) to check if the agent spots winning opportunities.

In [85]:
```
def trial_for_winning_opportunities(board):
    visualize(board)
    selected_move=alpha_beta_minimax(board)
    print("Selected Column:", selected_move)
    new_board= action_result(board, selected_move, 1)
    visualize(new_board)
    utility=terminal(new_board)

    if utility != None:
        print("The winner is Player:", utility)
    else:
        print("Alpha-Beta Pruning Minimax Algorithm could not find terminal state")
```

In [104…
```
board = np.array([[0, 0, 0, 0],
                  [0, 1, -1, 1],
                  [1, -1, -1, 1],
                  [-1, -1, 1, 1]])
%time trial_for_winning_opportunities(board)
```

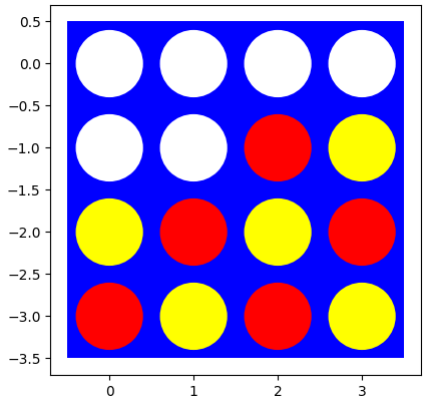



```
Selected Column: 3
```

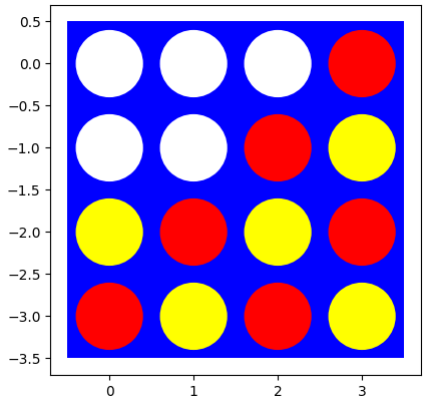



```
The winner is Player: 1
CPU times: total: 406 ms
Wall time: 397 ms
```

In [103…
```
board = np.array([[0, 0, 0, 0],
                  [0, 0, 1, -1],
                  [-1, 1, -1, 1],
                  [1, -1, 1, -1]])

%time trial_for_winning_opportunities(board)
```

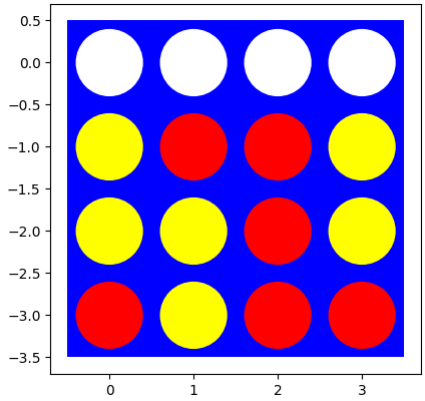



```
Selected Column: 3
```

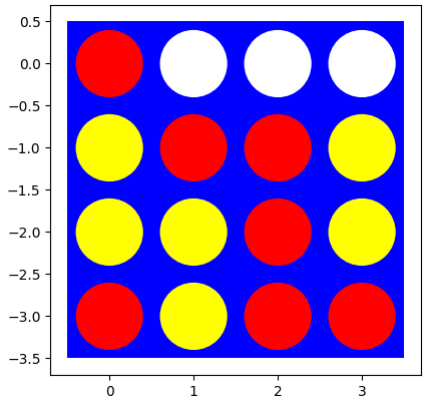


```
The winner is Player: 1
CPU times: total: 422 ms
Wall time: 441 ms
```

In [102…
```
board = np.array([[0, 0, 0, 0],
                  [-1, 1, 1, -1],
                  [-1, -1, 1, -1],
                  [1, -1, 1, 1]])

%time trial_for_winning_opportunities(board)
```

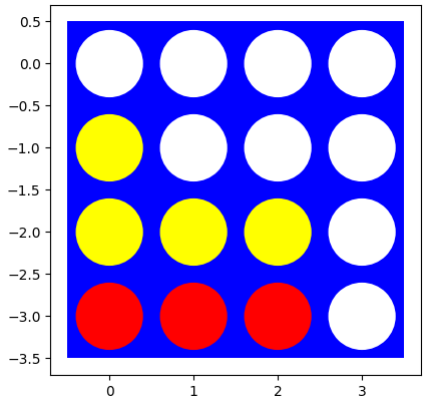


```
Selected Column: 0
```

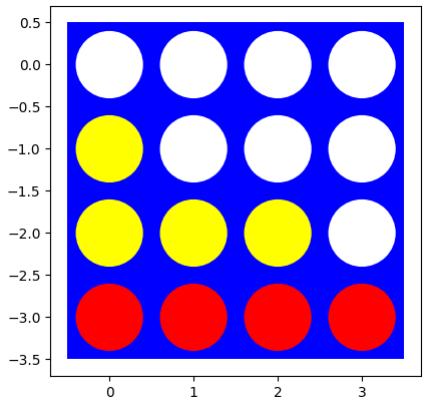


```
The winner is Player: 1
CPU times: total: 391 ms
Wall time: 380 ms
```

In [101…
```
board = np.array([[0, 0, 0, 0],
                  [-1, 0, 0, 0],
                  [-1, -1, -1, 0],
                  [1, 1, 1, 0]])

%time trial_for_winning_opportunities(board)
```



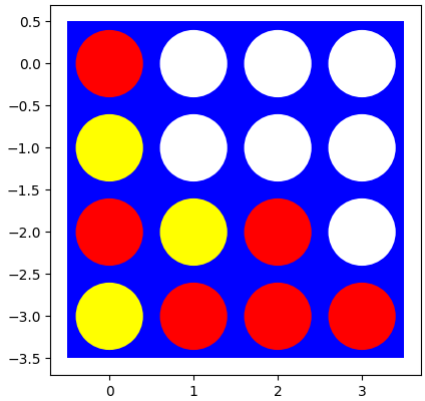
```
Selected Column: 3
```



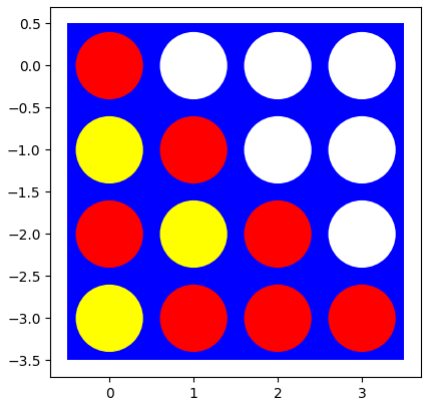
```
The winner is Player: 1
CPU times: total: 578 ms
Wall time: 560 ms
```

In [100…

```
board = np.array([[1, 0, 0, 0],
                  [-1, 0, 0, 0],
                  [ 1, -1, 1, 0],
                  [-1, 1, 1, 1]])

%time trial_for_winning_opportunities(board)
```

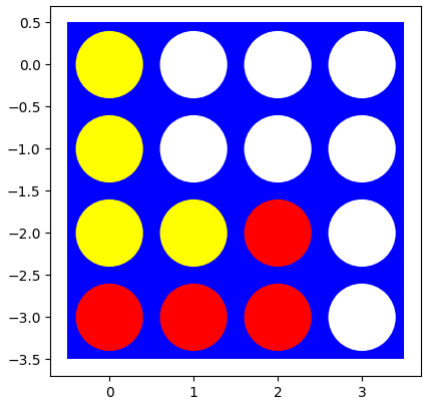


```
Selected Column: 1
```

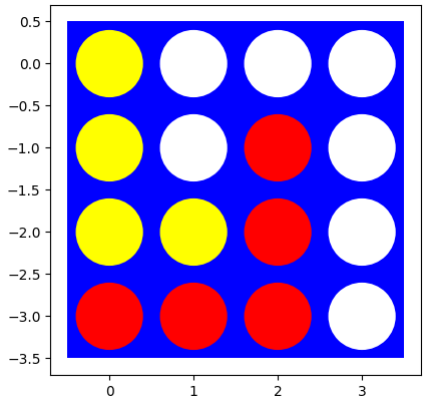


```
The winner is Player: 1
CPU times: total: 391 ms
Wall time: 390 ms
```

In [99]:
```
board = np.array([[-1, 0, 0, 0],
                  [-1, 0, 0, 0],
                  [-1, -1, 1, 0],
                  [1, 1, 1, 0]])

%time trial_for_winning_opportunities(board)
```



```
Selected Column: 2
```



```
Alpha-Beta Pruning Minimax Algorithm could not find terminal state
CPU times: total: 375 ms
Wall time: 379 ms
```

Discussion

Alpha-Beta Pruning Minimax Algorithm can detect the terminal state with some exceptions. 5 of the 6 boards above, it could find the shortest terminal state. Only the last one was the problem. The reason is it does not go to the shortest path but it goes to the first solution it finds.

After this move, there are 2 slots for terminal for player minimax. Even the opponent plays optimal, it will go to the solution. I wrote the possible two scenarios to show that minimax actually finds to solution.

How long does it take to make a move? Start with a smaller board with 4 columns and make the board larger by adding columns.

Game size is large. Thus, if I use an empty board with size of (7x6) to for the agent to make the first move, it takes take too much time compute all the possible nodes. I used half empty boards with different sizes.

In [182…
```
import time
time_list_minimax=[]
board = np.array([[0, 0, 0, 0],
                  [1, -1, 1, -1],
                  [-1, 1, -1, -1],
                  [1, 1, -1, 1]])
start_minimax = time.time()
display(alpha_beta_minimax(board, 11))
end_minimax = time.time()
time_list_minimax.append(end_minimax-start_minimax)


board = np.array([[0, 0, 0, 0, 0],
                  [-1, 1, -1, 1, -1],
                  [1, -1, 1, 1, -1],
                  [1, 1, -1, -1, 1]])
start_minimax = time.time()
display(alpha_beta_minimax(board, 13))
end_minimax = time.time()
time_list_minimax.append(end_minimax-start_minimax)


board = np.array([[0, 0, 0, 0, 0, 0],
                  [0, 1, -1, -1, -1, 1],
                  [1, -1, -1, 1, -1, -1],
                  [1, 1, -1, 1, -1, 1]])
start_minimax = time.time()
display(alpha_beta_minimax(board, 15))
end_minimax = time.time()
time_list_minimax.append(end_minimax-start_minimax)


board = np.array([[0, 0, 0, 0, 0, 0, 0],
                  [0, 0, 0, 0, 0, 0, 0],
                  [0, -1, -1, 1, 1, -1, -1],
                  [1, -1, 1, 1, 1, -1, 1]])
start_minimax = time.time()
display(alpha_beta_minimax(board, 21))
```

```
end_minimax = time.time()
time_list_minimax.append(end_minimax-start_minimax)
```

```
3
4
0
2
```

In [114…
```
time_list_minimax
```

Out[114]:
```
[0.003998756408691406,
 0.015990018844604492,
 0.0499918460845947,
 99.08339095115662]
```

Move ordering [5 points]

Starting the search with better moves will increase the efficiency of alpha-beta pruning. Describe and implement a simple move ordering strategy. Make a table that shows how the ordering strategies influence the time it takes to make a move?

4-connect is a solved problem. Throughout my search about the game online, I understand from experts that it makes sense to control the center for 4-connect. Thus, searching beginning from center to the edge is a approach. Thus, I defined the actions list accordingly, again.

In [183…
```
# all possible actions
def actions(board):
    actions = list()
    a = [0, 1, 2, 3, 4, 5, 6]
    #check if there is an available spot in that column
    for i in np.where(np.array(board[0]) == 0)[0]:
        actions.append(i)
    #move ordering
    actions = sorted(actions, key = lambda x: abs(x-actions[len(actions)//2]))
    return actions
```

Trying move ordering in dummy board

In [184…
```
board = [[0, 0, 0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 0, 0],
         [0, 0, 0, 1, 0, 0, 0],
         [0, 0, 0, 1, 0, 0, 0],
         [0,-1,-1, 1,-1, 0, 0]]
actions(board)
```

Out[184]:
```
[3, 2, 4, 1, 5, 0, 6]
```

In [185…
```
time_list_moveorder=[]
board = np.array([[0, 0, 0, 0],
                  [1, -1, 1, -1],
                  [-1, 1, -1, -1],
                  [1, 1, -1, 1]])
start_minimax = time.time()
display(alpha_beta_minimax(board, 11))
end_minimax = time.time()
time_list_moveorder.append(end_minimax-start_minimax)


board = np.array([[0, 0, 0, 0, 0],
                  [-1, 1, -1, 1, -1],
                  [1, -1, 1, 1, -1],
                  [1, 1, -1, -1, 1]])
start_minimax = time.time()
display(alpha_beta_minimax(board, 13))
end_minimax = time.time()
time_list_moveorder.append(end_minimax-start_minimax)


board = np.array([[0, 0, 0, 0, 0, 0, 0],
                  [0, 1, -1, -1, -1, 1],
                  [1, -1, -1, 1, -1, -1],
                  [1, 1, -1, 1, -1, 1]])
start_minimax = time.time()
display(alpha_beta_minimax(board, 15))
end_minimax = time.time()
time_list_moveorder.append(end_minimax-start_minimax)


board = np.array([[0, 0, 0, 0, 0, 0, 0],
                  [0, 0, 0, 0, 0, 0, 0],
                  [0, -1, -1, 1, 1, -1, -1],
                  [1, -1, 1, 1, 1, -1, 1]])
start_minimax = time.time()
display(alpha_beta_minimax(board, 21))
end_minimax = time.time()
time_list_moveorder.append(end_minimax-start_minimax)
time_list_moveorder
```

```
3
4
3
3
```

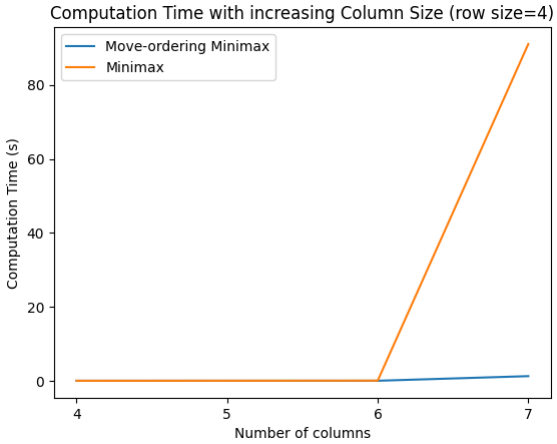
Out[185]:
```
[0.0040149688720703125,
 0.010989904403686523,
 0.00997781753540039,
 1.2562930583953857]
```

In [186…
```
print('Computation times with move ordering', time_list_moveorder)
print('Computation times without move ordering', time_list_minimax)
```

```
Computation times with move ordering [0.0040149688720703125, 0.010989904403686523, 0.00997781753540039, 1.2562930583953857]
Computation times without move ordering [0.0059773921966552734, 0.01699376106262207, 0.04498696327209473, 91.07605600357056]
```

In [187…
```
plt.plot([4,5,6,7], time_list_moveorder, label="Move-ordering Minimax")
plt.plot([4,5,6,7], time_list_minimax, label="Minimax")
plt.xticks(range(4,7+1))
plt.title('Computation Time with increasing Column Size (row size=4)')
plt.xlabel("Number of columns")
plt.ylabel("Computation Time (s)")
plt.legend(loc="best")
```

Out[187]:
```
<matplotlib.legend.Legend at 0x16a814d4b80>
```



Defining the move ordering strategy improved the computation time. The difference increases dramatically with the increasing board size as expected.

The first few moves [5 points]

Start with an empty board. This is the worst case scenario for minimax search since it needs solve all possible games that can be played (minus some pruning) before making the decision. What can you do?

As explained in the move ordering part, taking the control of center is very important in this game. Thus, when opening the game, I will make the first move as always to the center.

In [188…
```
actions(board)[0]
```

Out[188]:
```
3
```

In [189…
```
def alpha_beta_minimax(board, limit_moves = +math.inf, player = 1):
    deep = 0
    if np.count_nonzero(board, axis=None)<=2:
        move=actions(board)[0]
    else:
        value, move = max_value_alpha_beta(limit_moves, deep, board, player, -math.inf, +math.inf)
    return move
```

In [190…
```
board = np.array([[0, 0, 0, 0, 0],
                  [0, 0, 0, 0, 0],
                  [0, 0, 0, 0, 0],
                  [0, 0, -1, 0, 0]])
actions(board)
print("Selected Action is ", alpha_beta_minimax(board))
```

```
Selected Action is  2
```

Try with Empty Board

In [191…
```
board = np.array([[0, 0, 0, 0, 0, 0, 0],
                  [0, 0, 0, 0, 0, 0, 0],
                  [0, 0, 0, 0, 0, 0, 0],
                  [0, 0, 0, 0, 0, 0, 0],
                  [0, 0, 0, 0, 0, 0, 0],
                  [0, 0, 0, 0, 0, 0, 0]])

actions(board)
%time  print("Selected Action is ", alpha_beta_minimax(board))
```

```
Selected Action is  3
CPU times: total: 0 ns
Wall time: 0 ns
```

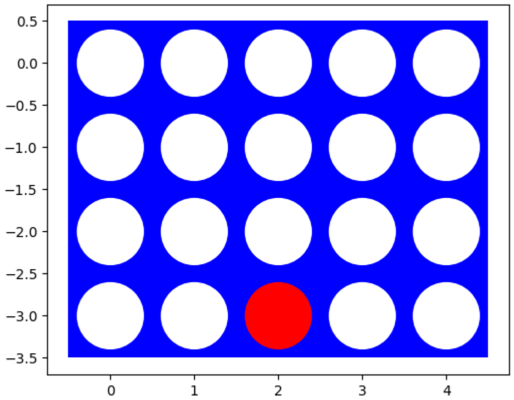
Playtime [5 points]

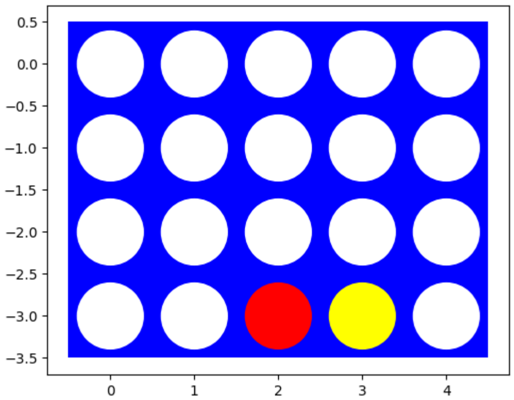
Let the Minimax Search agent play a random agent on a small board. Analyze wins, losses and draws.

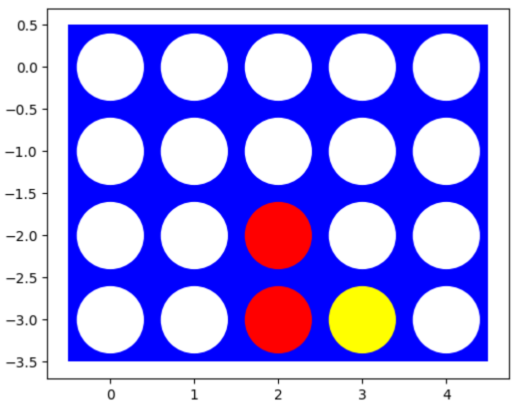
In [132…
```
# Your code/ answer goes here.
empty_board(shape=(4, 4))
```

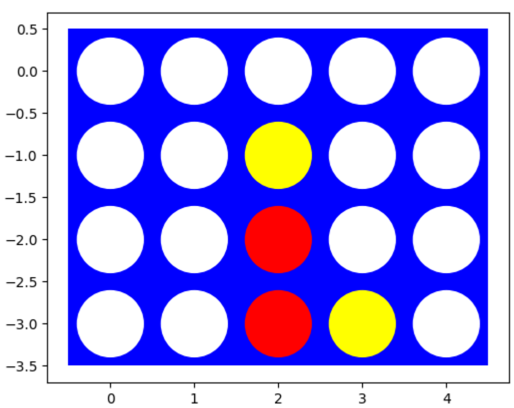
Out[132]:
```
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]])
```

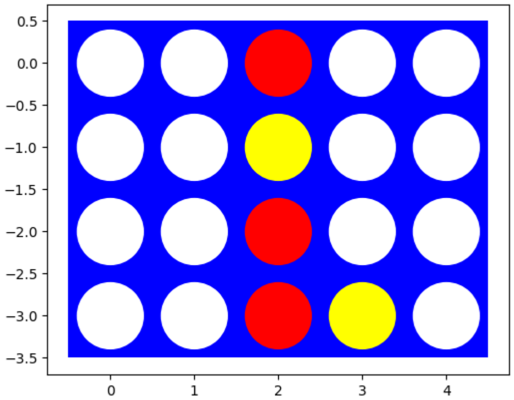
In [133…
```
first_player=1#First Player 1
result_random_vs_random=pd.DataFrame([game(alpha_beta_minimax,random_player, first_player, True,1, 4,5)])
print(result_random_vs_random)

result_arr = result_random_vs_random.to_numpy()
utility_arr=([1, -1, 0])
plt.bar(utility_arr,result_arr[0])
plt.xlabel("Utility")
plt.ylabel("Winning Frequency")
plt.title("Result Table for 1 Games between Minimax and Random. First Player:"+str(first_player))
```

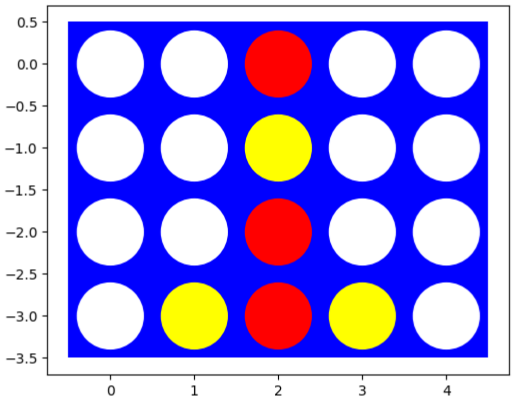

0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0
1
2
3
4

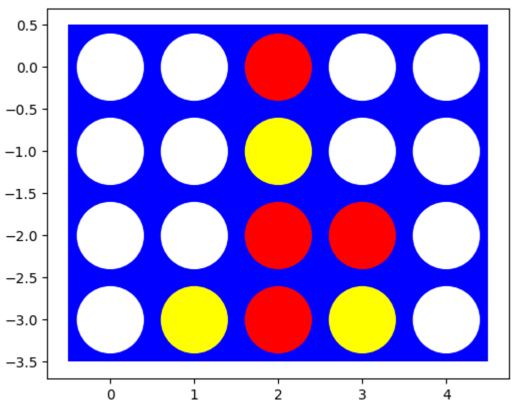

0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0
1
2
3
4

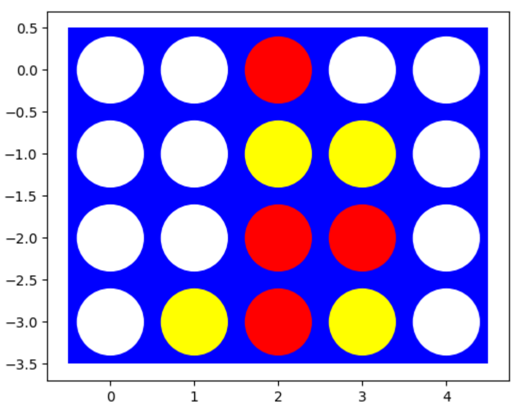

0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0
1
2
3
4

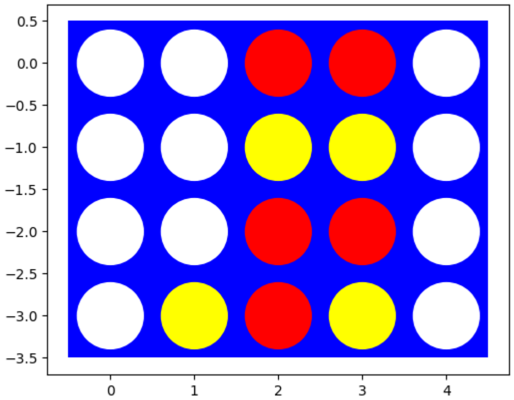

0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0
1
2
3
4

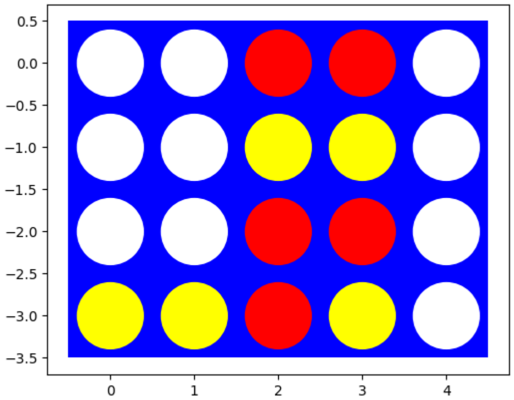

0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0
1
2
3
4

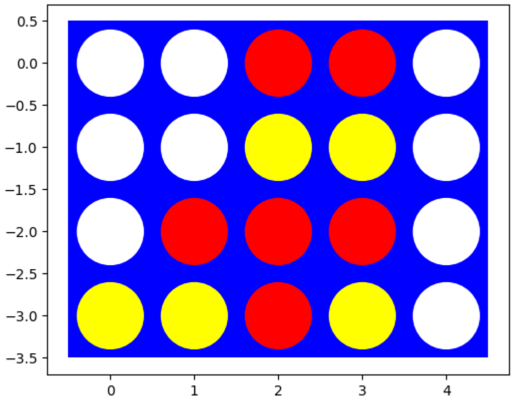

0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0
1
2
3
4

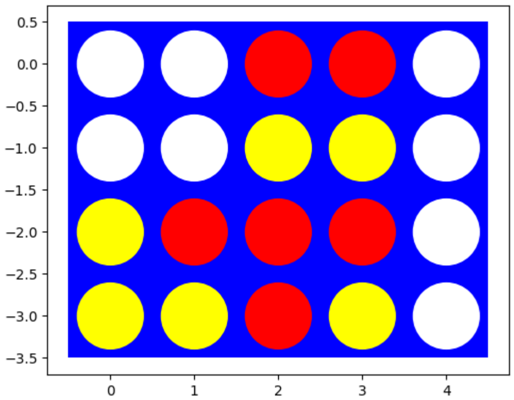

0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0
1
2
3
4

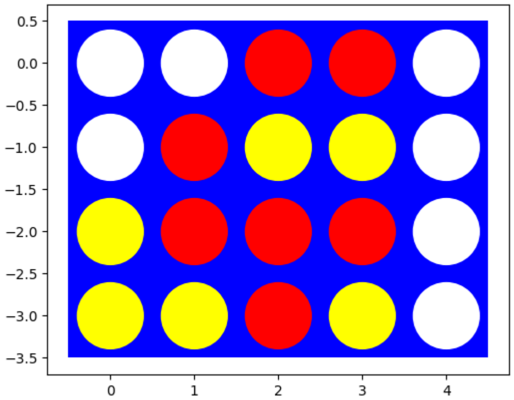

0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0
1
2
3
4

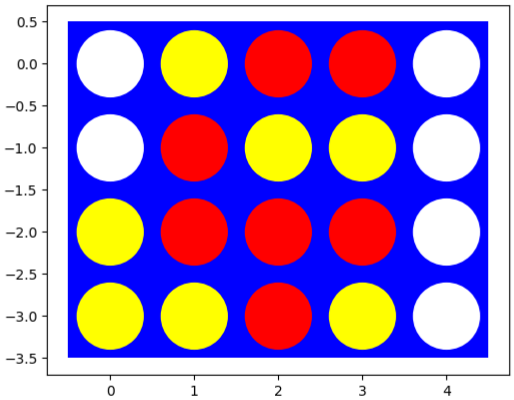

0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0
1
2
3
4

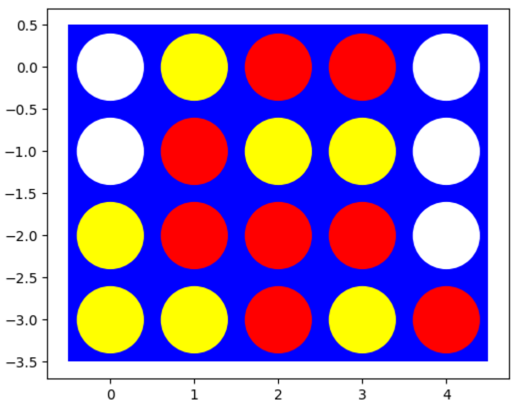

0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0
1
2
3
4

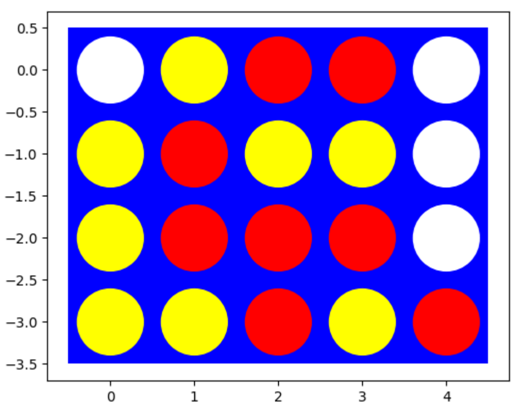

0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0
1
2
3
4

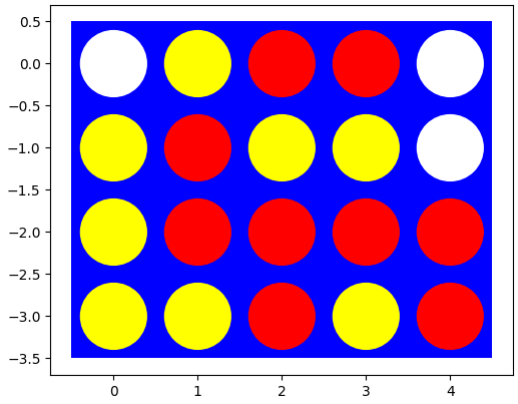

0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0
1
2
3
4


0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0 1 2 3 4


0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0 1 2 3 4

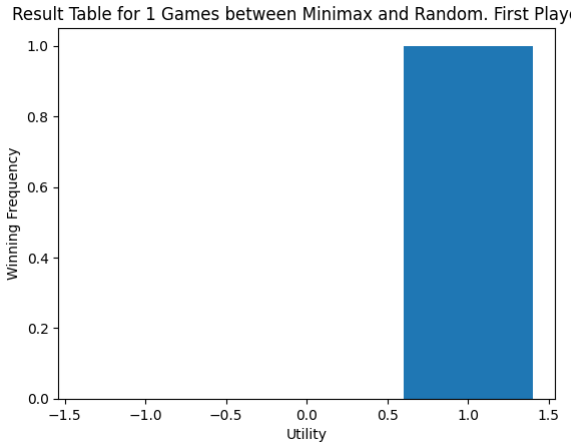

0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0 1 2 3 4


0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0 1 2 3 4



```
   1  -1   0
0  1   0   0
```

Out[133]: Text(0.5, 1.0, 'Result Table for 1 Games between Minimax and Random. First Player:1')

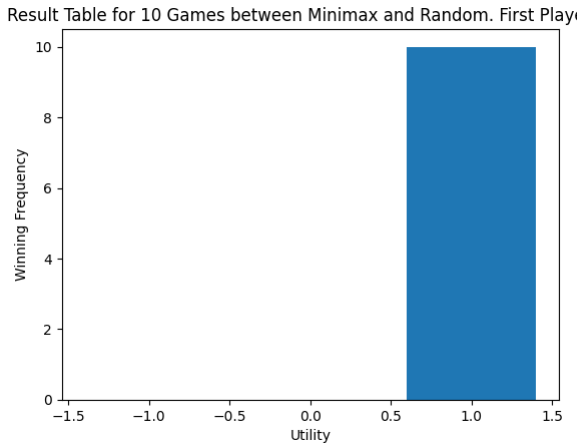


In [111…

```
first_player=-1#First Player -1
result_random_vs_random=pd.DataFrame([game(alpha_beta_minimax,random_player, first_player, False,10, 4,5)])
print(result_random_vs_random)

result_arr = result_random_vs_random.to_numpy()
utility_arr=([1, -1, 0])
plt.bar(utility_arr,result_arr[0])
plt.xlabel("Utility")
plt.ylabel("Winning Frequency")
plt.title("Result Table for 10 Games between Minimax and Random. First Player:"+str(first_player))
```

```
   1  -1   0
0  10   0   0
```

Out[111]: Text(0.5, 1.0, 'Result Table for 10 Games between Minimax and Random. First Player:-1')



For 4x5 board, Minimax always wins even though random starts first.

Task 4: Heuristic Alpha-Beta Tree Search

Heuristic evaluation function [15 points]

Define and implement a heuristic evaluation function.

Please check the code and the explanations inside of it for the heuristic function

In [219…

```
def heuristic_evaluation(state, heur_id):
    # check if it is the terminal state.
    state=state.copy()
    value = terminal(state)
```

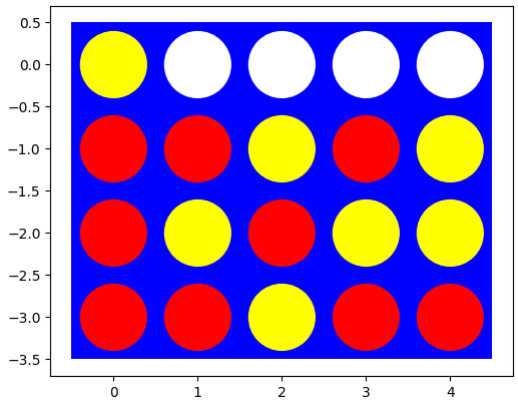
```
    if value is not None:
        return value, True

    if heur_id == 1:
        points_3row = 0.9
        points_2row = 0.6
        points_1row = 0.3

    elif heur_id == 2:
        points_3row = 0.6
        points_2row = 0.4
        points_1row = 0.2

    elif heur_id == 3:
        points_3row = 30
        points_2row = 20
        points_1row = 10


    point = 0
    h = len(state)
    w = len(state[0])


    # this part looks at horizontal groups of 4
    # elements and check if the sum of them is 3
    # or -3.

    # The only way to have sum of three is having
    # empty slot (0) once and disk 1 or -1 three times.
    # Thus, I did not check any exceptions.


    for i in range(h):
        for j in range(w - 3):

            if(state[i][j] + state[i][j + 1] + state[i][j + 2] + state[i][j + 3] == -3):
                point -= points_3row
                return point, False

    for i in range(h):
        for j in range(w - 3):

            if(state[i][j] + state[i][j + 1] + state[i][j + 2] + state[i][j + 3] == 3):
                point += points_3row
                return point, False




    # this part looks at vertical groups of 4
    # elements and check if the sum of them is 3
    # or -3.

    # The only way to have sum of three is having
    # empty slot (0) once and disk 1 or -1 three times.
    # Thus, I did not check any exceptions.

    for j in range(w):
        for i in range(h - 3):

            if(state[i][j] + state[i + 1][j] + state[i + 2][j] + state[i + 3][j] == -3):
                point -= points_3row
                #print("aaaaa")
                return point, False

    for j in range(w):
        for i in range(h - 3):

            if(state[i][j] + state[i + 1][j] + state[i + 2][j] + state[i + 3][j] == 3):
                #print("aaaaa")
                point += points_3row
                return point, False

    # this part looks at diagonal groups of 4
    # elements and check if the sum of them is 3
    # or -3.

    # The only way to have sum of three is having
    # empty slot (0) once and disk 1 or -1 three times.
    # Thus, I did not check any exceptions.

    for i in range(h - 3):
        for j in range(w - 3):

            if(state[i][j] + state[i + 1][j + 1] + state[i + 2][j + 2] + state[i + 3][j + 3] == -3):
                point -= points_3row
                return point, False


            if(state[i + 3][j] + state[i + 2][j + 1] + state[i + 1][j + 2] + state[i][j + 3] == -3):
                point -= points_3row
                return point, False


    for i in range(h - 3):
        for j in range(w - 3):
            if(state[i][j] + state[i + 1][j + 1] + state[i + 2][j + 2] + state[i + 3][j + 3] == 3):
                point += points_3row
                return point, False

            if(state[i + 3][j] + state[i + 2][j + 1] + state[i + 1][j + 2] + state[i][j + 3] == 3):
                point += points_3row
                return point, False

    # this part looks at horizontal groups of 4
    # elements and check if the sum of them is 2
    # or -2.

    #There are some exceptions for this condition:
    # Having anothercolor disk.

    # 1 1 -1 1 has also sum of 2. However,
    # there is not a winning opportunity here.
    # Thus, we should exclude this.
```

```
    # Multiply with -1 for opponent's situation.

    for i in range(h):
        for j in range(w - 3):
            horizontal_array_4 = [state[i][j], state[i][j + 1], state[i][j + 2], state[i][j + 3]]
            sum_hor = state[i][j] + state[i][j + 1] + state[i][j + 2] + state[i][j + 3]

            if(sum_hor == 2 and -1 not in horizontal_array_4):
                point += points_2row
            if(sum_hor == -2 and 1 not in horizontal_array_4):
                point -= points_2row

    # this part looks at vertical groups of 4
    # elements and check if the sum of them is 2
    # or -2.
    #There are some exceptions for this condition:
    # Having anothercolor disk.

    # 1 1 -1 1 has also sum of 2. However,
    # there is not a winning opportunity here.
    # Thus, we should exclude this.
    # Multiply with -1 for opponent's situation.


    for j in range(w):
        for i in range(h - 3):
            vertical_array_2 = [state[i][j], state[i+1][j], state[i+2][j], state[i+3][j]]
            sum_ver= state[i][j] + state[i + 1][j] + state[i + 2][j] + state[i + 3][j]

            if(sum_ver == 2 and -1 not in vertical_array_2):
                point += points_2row
            if(sum_ver == -2 and 1 not in vertical_array_2):
                point -= points_2row


    # this part looks at diagonal groups of 4
    # elements and check if the sum of them is 2
    # or -2.

    # The only way to have sum of three is having
    # empty slot (0) once and disk 1 or -1 three times.
    # Thus, I did not check any exceptions.

    for i in range(h - 3):
        for j in range(w - 3):
            diagonal1 = [state[i][j], state[i + 1][j + 1], state[i + 2][j + 2], state[i + 3][j + 3]]
            diagonal2 = [state[i + 3][j], state[i + 2][j + 1], state[i + 1][j + 2], state[i][j + 3]]

            sumdiagonal1 = state[i][j] + state[i + 1][j + 1] + state[i + 2][j + 2] + state[i + 3][j + 3]
            sumdiagonal2 = state[i + 3][j] + state[i + 2][j + 1] + state[i + 1][j + 2] + state[i][j + 3]

            if(sumdiagonal1 == 2 and -1 not in diagonal1):
                point += points_2row
            if(sumdiagonal1 == -2 and 1 not in diagonal1):
                point -= points_2row
            if(sumdiagonal2 == 2 and -1 not in diagonal2):
                point += points_2row
            if(sumdiagonal2 == -2 and 1 not in diagonal2):
                point -= points_2row


    # this part looks at horizontal groups of 4
    # elements and check if the sum of them is 1
    # or -1.

    for i in range(h):
        for j in range(w - 3):
            horizontal_array_1 = [state[i][j], state[i][j + 1], state[i][j + 2], state[i][j + 3]]
            sum_hor = state[i][j] + state[i][j + 1] + state[i][j + 2] + state[i][j + 3]

            if(sum_hor == 1 and -1 not in horizontal_array_1):
                point += points_1row
            if(sum_hor == -1 and 1 not in horizontal_array_1):
                point -= points_1row
    # this part looks at vertical groups of 4
    # elements and check if the sum of them is 1
    # or -1.

    for j in range(w):
        for i in range(h - 3):
            vertical_array_4 = [state[i][j], state[i+1][j], state[i+2][j], state[i+3][j]]
            sum_ver= state[i][j] + state[i + 1][j] + state[i + 2][j] + state[i + 3][j]

            if(sum_ver == 1 and -1 not in vertical_array_4):
                point += points_1row
            if(sum_ver == -1 and 1 not in vertical_array_4):
                point -= points_1row
    # this part looks at diagonal groups of 4
    # elements and check if the sum of them is 1
    # or -1.

    for i in range(h - 3):
        for j in range(w - 3):
            diagonal1 = [state[i][j], state[i + 1][j + 1], state[i + 2][j + 2], state[i + 3][j + 3]]
            diagonal2 = [state[i + 3][j], state[i + 2][j + 1], state[i + 1][j + 2], state[i][j + 3]]

            sumdiagonal1 = state[i][j] + state[i + 1][j + 1] + state[i + 2][j + 2] + state[i + 3][j + 3]
            sumdiagonal2 = state[i + 3][j] + state[i + 2][j + 1] + state[i + 1][j + 2] + state[i][j + 3]

            if(sumdiagonal1 == 1 and -1 not in diagonal1):
                point += points_1row
            if(sumdiagonal1 == -1 and 1 not in diagonal1):
                point -= points_1row
            if(sumdiagonal2 == 1 and -1 not in diagonal2):
                point += points_1row
            if(sumdiagonal2 == -1 and 1 not in diagonal2):
                point -= points_1row


    return point, False
```

Trying the heuristic function with different boards

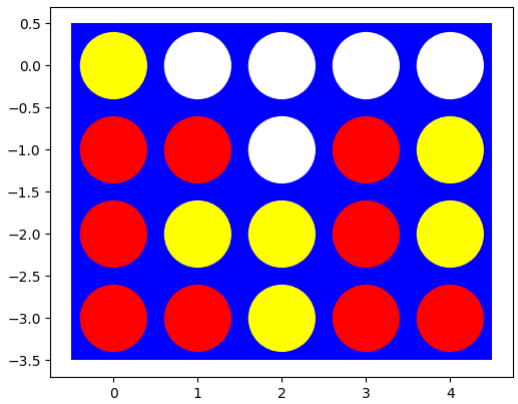
```
In [135… board = np.array([[-1, 0, 0, 0, 0],
```

```
                [1, 1, -1, 1, -1],
                [1, -1, 1, -1, -1],
                [1, 1, -1, 1, 1]])
visualize(board)
heuristic_evaluation(board, 1)
```



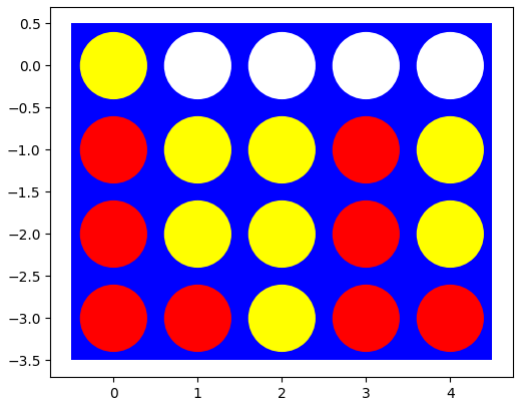
Out[135]: (0.9, False)

In [136…
```
board = np.array([[-1, 0, 0, 0, 0],
                [1, 1, 0, 1, -1],
                [1, -1, -1, 1, -1],
                [1, 1, -1, 1, 1]])
visualize(board)
heuristic_evaluation(board, 1)
```



Out[136]: (0.9, False)

In [137…
```
board = np.array([[-1, 0, 0, 0, 0],
                [1, -1, -1, 1, -1],
                [1, -1, -1, 1, -1],
                [1, 1, -1, 1, 1]])
visualize(board)
heuristic_evaluation(board, 1)
```



Out[137]: (-0.9, False)

Cutting off search [10 points]

Modify your Minimax Search with Alpha-Beta Pruning to cut off search at a specified depth and use the heuristic evaluation function. Experiment with different cutoff values.

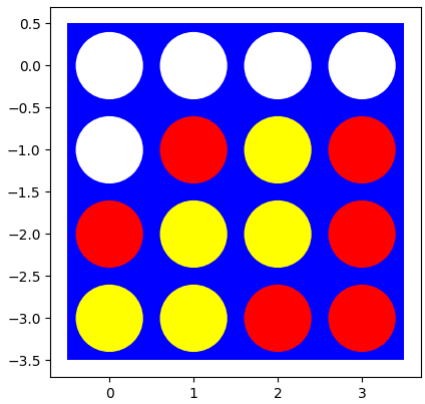
In [237…
```
def heuristic_alpha_beta(state, cutoff_depth, player = 1, mode = 1):
    if np.count_nonzero(state, axis=None)<=2:
        move=actions(state)[0]
    else:
        value, move = max_value_alpha_beta(state, player, -math.inf, +math.inf, 0, cutoff_depth, mode)
    return move

def max_value_alpha_beta(state, player, alpha, beta, search_depth, cutoff_depth, mode):
    value, terminal_eval = heuristic_evaluation(state, mode)
```

```
    if((cutoff_depth is not None and search_depth >= cutoff_depth) or terminal_eval):
        if(terminal_eval):
            alpha, beta = value, value
        return value, None

    value, move = -math.inf, None

    for action in actions(state):
        val2, act2 = min_value_alpha_beta(action_result(state, action, player), player, alpha, beta, search_depth + 1, cutoff_depth, mode)
        if val2 > value:
            value, move = val2, action
            alpha = max(alpha, value)
        if value >= beta: return value, move

    return value, move

def min_value_alpha_beta(state, player, alpha, beta, search_depth, cutoff_depth, mode):
    value, terminal_eval = heuristic_evaluation(state, mode)
    if((cutoff_depth is not None and search_depth >= cutoff_depth) or terminal_eval):
        if(terminal_eval):
            alpha, beta = value, value
        return value, None

    value, move = +math.inf, None

    # check all the actions options in the current state
    # define beta and alpha again
    # update with the smallest value

    for action in actions(state):
        val2, act2 = max_value_alpha_beta(action_result(state, action, switch(player)), player, alpha, beta, search_depth + 1, cutoff_depth, mode)
        if val2 < value:
            value, move = val2, action
            beta = min(beta, value)
        if value <= alpha: return value, move

    return value, move

def trial_for_winning_opportunities(state, cutoff_depth):
    visualize(state)
    selected_move=heuristic_alpha_beta(state, cutoff_depth)
    print("Selected Column:", selected_move)
    new_state= action_result(state, selected_move, 1)
    visualize(new_state)
    utility=terminal(new_state)

    if utility != None:
        print("The winner is Player:", utility)
    else:
        print("Alpha-Beta Pruning Minimax Algorithm could not find terminal state")
```

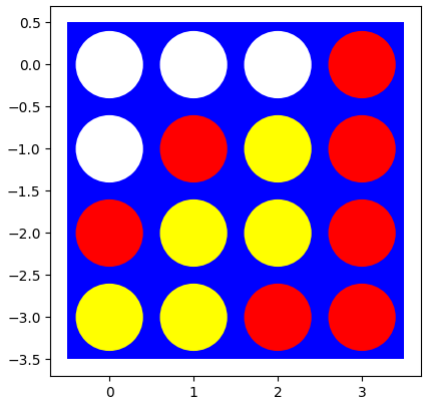
Experiment with the same manually created boards as above to check if the agent spots wining opportunities.

In [139…
```
board = np.array([[0, 0, 0, 0],
                  [0, 1, -1, 1],
                  [1, -1, -1, 1],
                  [-1, -1, 1, 1]])
trial_for_winning_opportunities(board, 5)
```

First Board

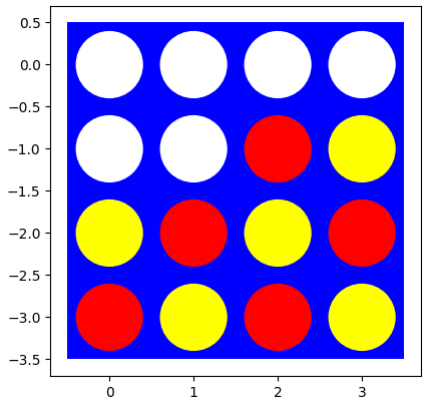


Board after selected move

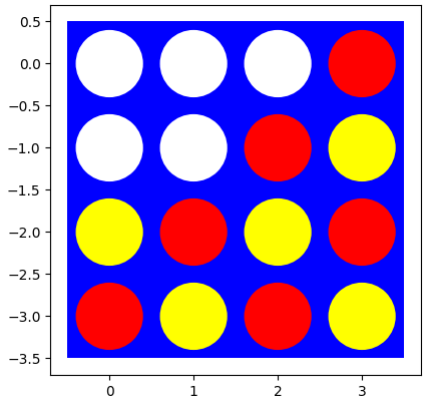


In [142…
```
board = np.array([[0, 0, 0, 0],
                  [0, 0, 1, -1],
                  [-1, 1, -1, 1],
```

```
                [1, -1, 1, -1]])
trial_for_winning_opportunities(board, 5)
```



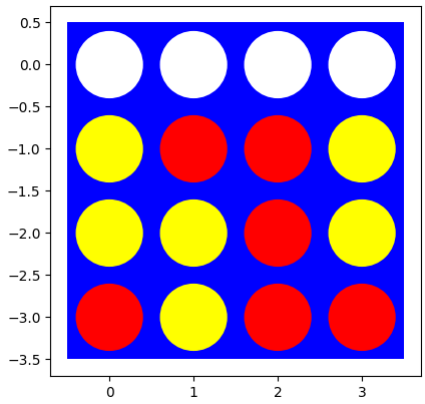
```
Selected Column: 3
```



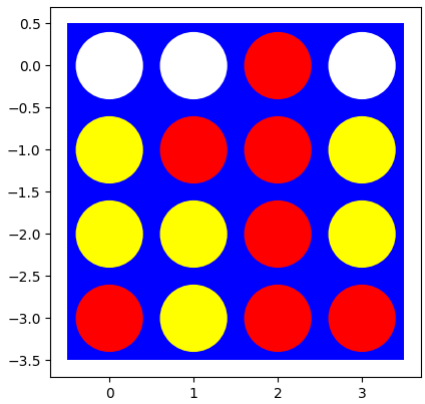
```
The winner is Player: 1
```

In [144…

```
board = np.array([[0, 0, 0, 0],
                  [-1, 1, 1, -1],
                  [-1, -1, 1, -1],
                  [1, -1, 1, 1]])

%time trial_for_winning_opportunities(board, 5)
```

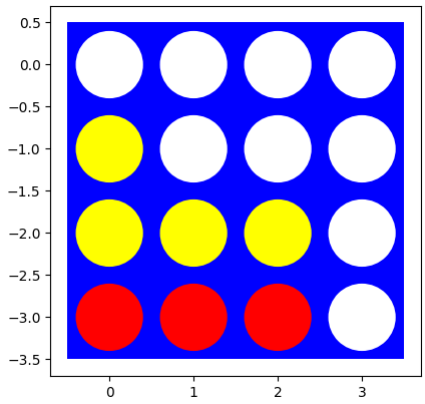


```
Selected Column: 2
```

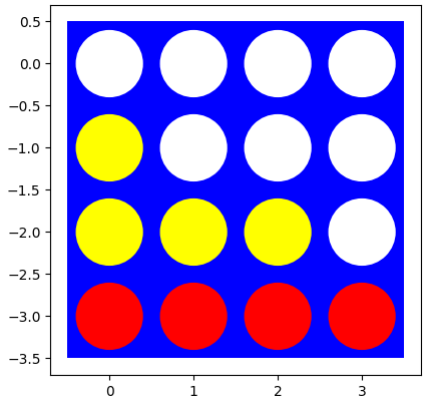


```
The winner is Player: 1
CPU times: total: 406 ms
Wall time: 436 ms
```

In [145…
```
board = np.array([[0, 0, 0, 0],
                  [-1, 0, 0, 0],
                  [-1, -1, -1, 0],
                  [1, 1, 1, 0]])

%time trial_for_winning_opportunities(board,5)
```

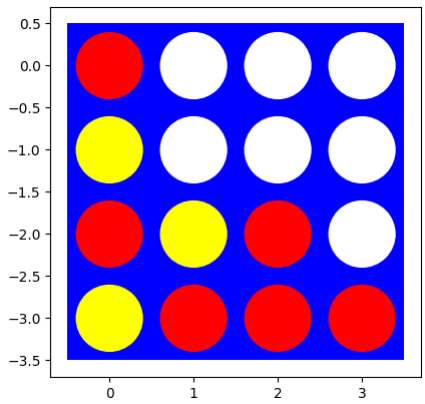



```
Selected Column: 3
```

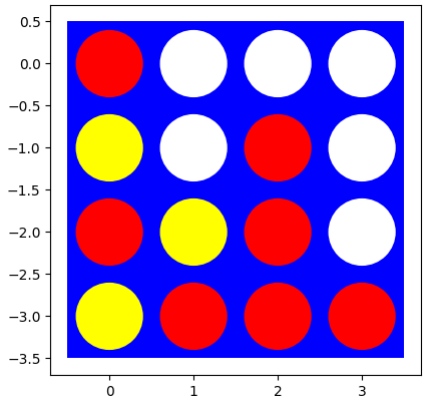



```
The winner is Player: 1
CPU times: total: 391 ms
Wall time: 385 ms
```

In [149…
```
board = np.array([[1, 0, 0, 0],
                  [-1, 0, 0, 0],
                  [ 1, -1, 1, 0],
                  [-1, 1, 1, 1]])

%time trial_for_winning_opportunities(board,42)
```




```
Selected Column: 2
```



```
Alpha-Beta Pruning Minimax Algorithm could not find terminal state
CPU times: total: 406 ms
Wall time: 390 ms
```

This is an interesting case. Move ordering checks the column 2 first.

After this move, there are 2 different terminal opportunities for red.

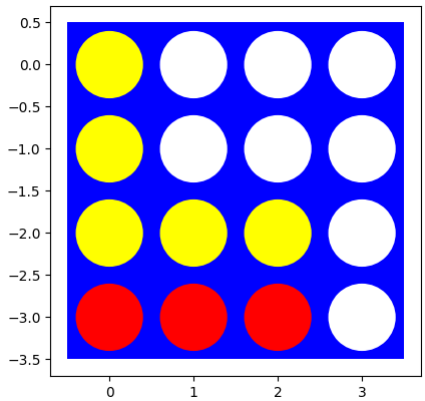
Whatever yellow does, red will win in the next round.

Since the algorithm begins from here and finds the solution, it gives the first solution.

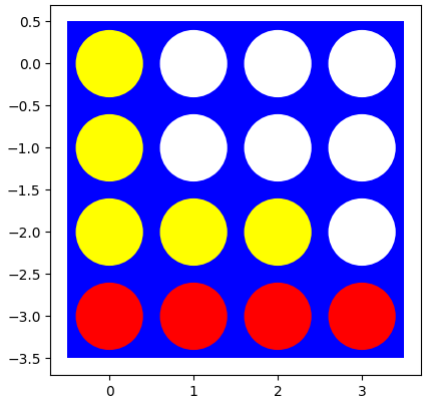
For the exact same board, minimax without move-ordering gives the shortest solution since it begins searching from column 0.

I checked the heristic values. They are correct. Because of the pruning, it still tends to go to the first solution it finds. After this move, whatever opponent does, minimax player will win. Maybe for the action for terminal cases, I can write another condition before beginning to minimax like I did for the opening 2 moves (It does not go through minimax, just puts the disk to the center.).

In [151…
```
board = np.array([[-1, 0, 0, 0],
                  [-1, 0, 0, 0],
                  [-1, -1, -1, 0],
                  [1, 1, 1, 0]])
%time trial_for_winning_opportunities(board,5)
```

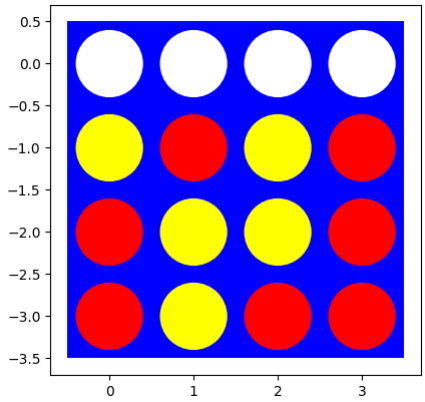


```
Selected Column: 3
```

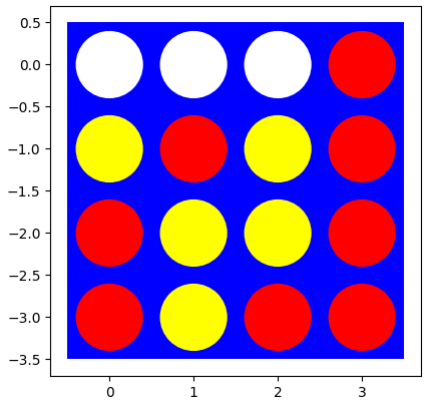


```
The winner is Player: 1
CPU times: total: 391 ms
Wall time: 392 ms
```

In [152…
```
board = np.array([[0, 0, 0, 0],
                  [-1, 1, -1, 1],
                  [1, -1, -1, 1],
                  [1, -1, 1, 1]])
%time trial_for_winning_opportunities(board,5)
```



```
Selected Column: 3
```

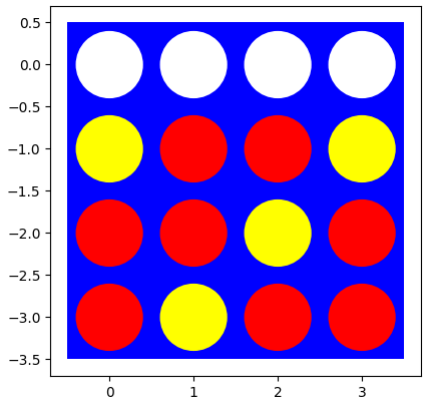


```
The winner is Player: 1
CPU times: total: 391 ms
Wall time: 384 ms
```

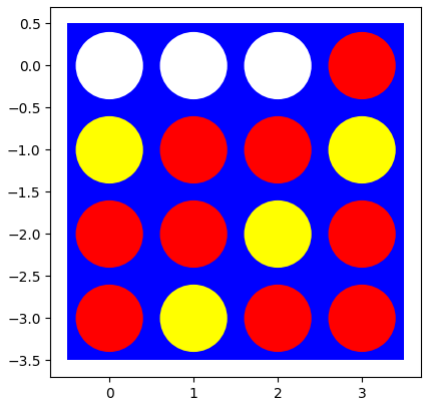
How long does it take to make a move? Start with a smaller board with 4 columns and make the board larger by adding columns.

In [153…

```
board = np.array([[0, 0, 0, 0],
                  [-1, 1, 1, -1],
                  [1, 1, -1, 1],
                  [1, -1, 1, 1]])
%time trial_for_winning_opportunities(board,5)
```



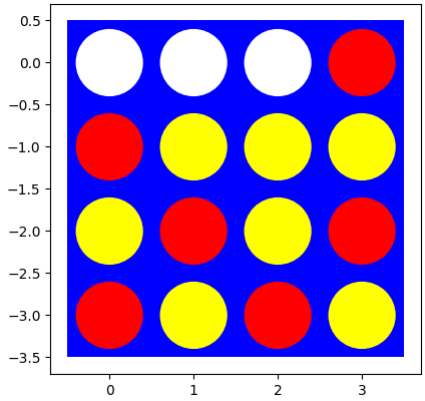
```
Selected Column: 3
```



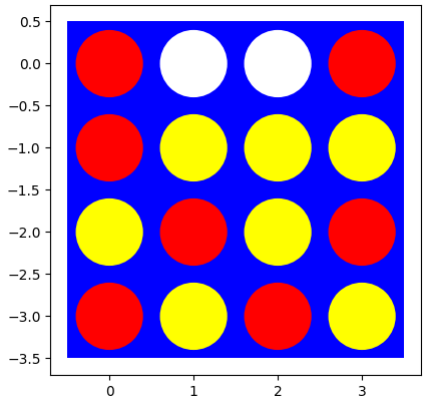
```
The winner is Player: 1
CPU times: total: 391 ms
Wall time: 445 ms
```

Let's check if the algorith can also successfully block the opponents terminal.

In [154…
```
board = np.array([[0, 0, 0, 1],
                  [1, -1, -1, -1],
                  [-1, 1, -1, 1],
                  [1, -1, 1, -1]])
%time trial_for_winning_opportunities(board,5)
```




```
Selected Column: 0
```




```
Alpha-Beta Pruning Minimax Algorithm could not find terminal state
CPU times: total: 375 ms
Wall time: 384 ms
```

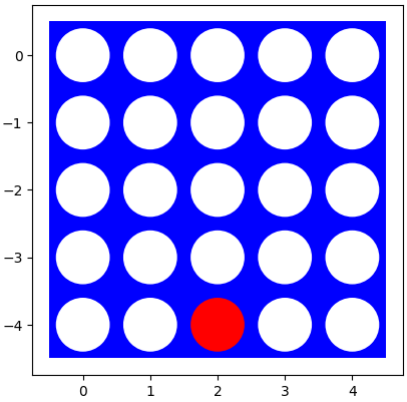
Yes, it can block the opponent's terminal!

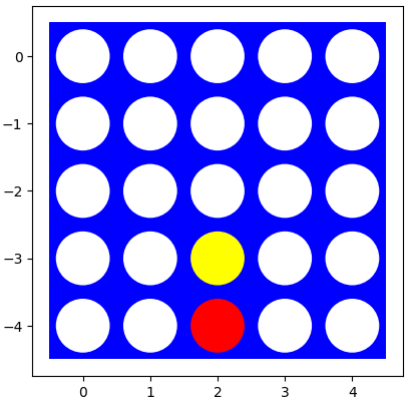
Playtime [5 points]

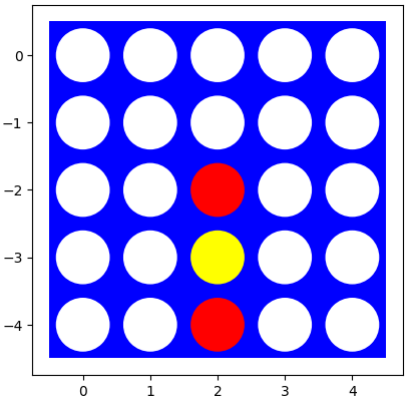
Let two heuristic search agents (different cutoff depth, different heuristic evaluation function) compete against each other on a reasonably sized board. Since there is no randomness, you only need to let them play once.

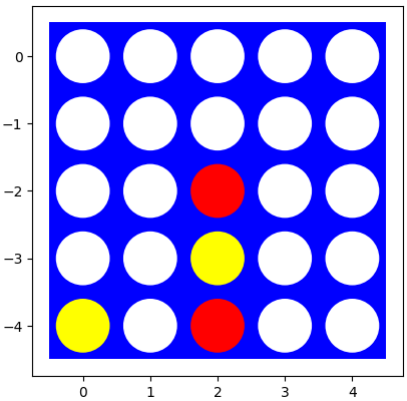
In [236…
```
def switch_heur(player):
    if player == 1:
        return -1, 2, 5
    else:
        return 1, 1, 10


def game_heur(player1, first_player, Verbose, n = 1 , h=4, w=4):
    result_record = {1: 0, -1: 0, 0: 0}
    if first_player==1:
        cutoff=10
        mode=1
    else:
        cutoff=5
        mode=2
    #n plays between 2 players
    for i in range(n):
        iters = 0
        board = empty_board(shape=(h, w))
        w=len(board[0])
        h=len(board)
        player=first_player
        current_player = player1

        #iteration during 1 game
        while iters < w*h+1:
            iters += 1

            action = heuristic_alpha_beta(board, cutoff, player, mode)

            board = action_result(board, action, player)
            if Verbose==True:
                visualize(board)
            winner = terminal(board)

            if winner != None:
                result_record[winner] += 1
                print("winner is player:", winner)
                break

            player, mode,  cutoff = switch_heur(player)

    return result_record
```

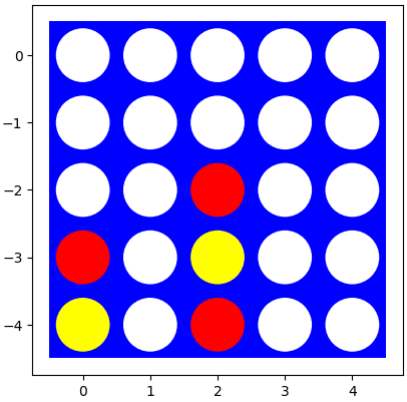
In [60]:
```
first_player=1#First Player 1
result_heur_vs_heur=pd.DataFrame([game_heur(random_player, first_player, True, n=1, h=5, w=5)])
print(result_heur_vs_heur)

result_arr = result_heur_vs_heur.to_numpy()
utility_arr=([1, -1, 0])
plt.bar(utility_arr,result_arr[0])
plt.xlabel("Utility")
plt.ylabel("Winning Frequency")
plt.title("Result Table for 1 Game between 2 Heur Players. First Player:"+str(first_player))
```

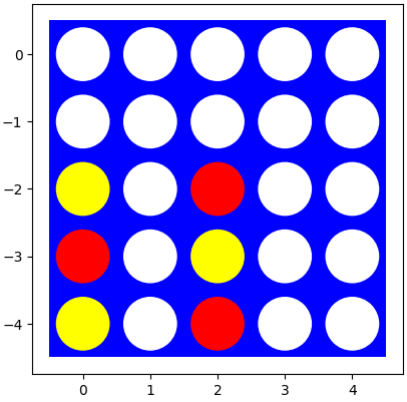


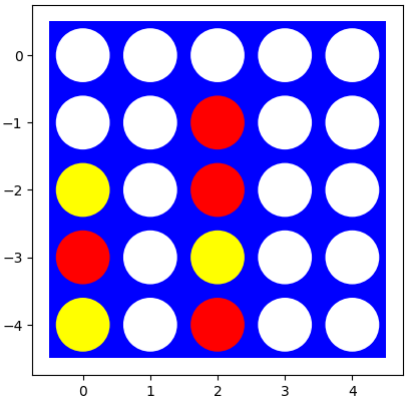


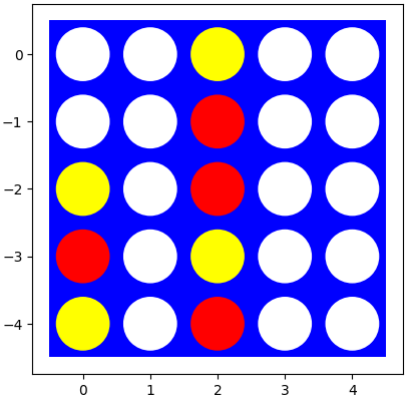


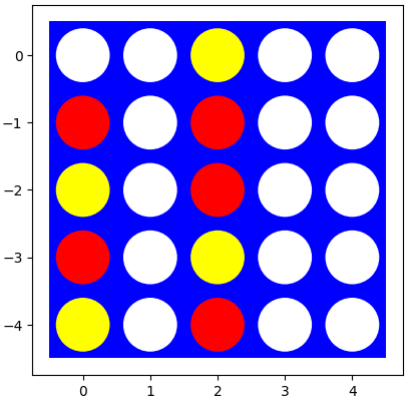


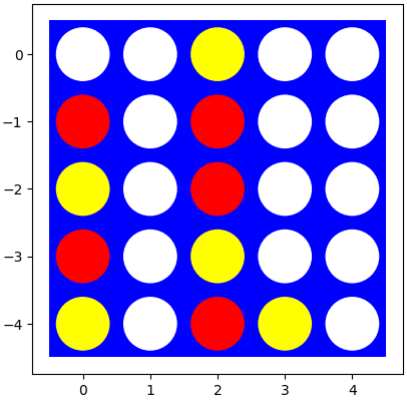

0
−1
−2
−3
−4
0
1
2
3
4

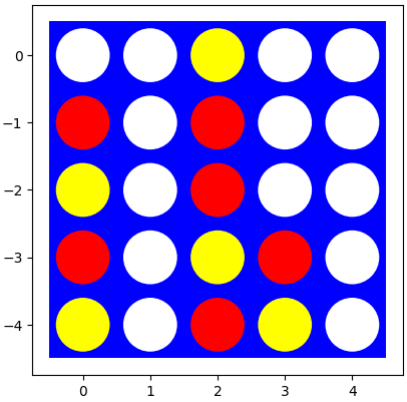

0
−1
−2
−3
−4
0
1
2
3
4

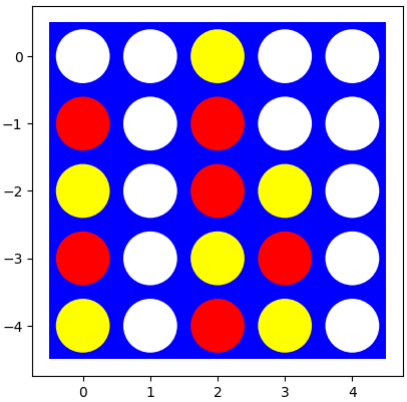

0
−1
−2
−3
−4
0
1
2
3
4

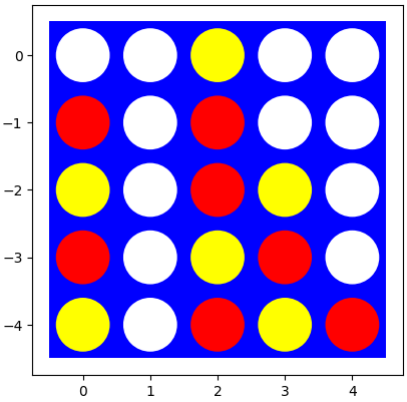

0
−1
−2
−3
−4
0
1
2
3
4

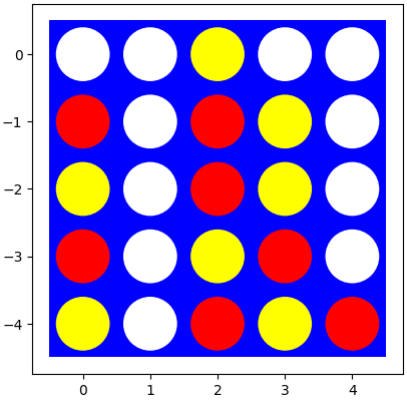

0
−1
−2
−3
−4
0
1
2
3
4

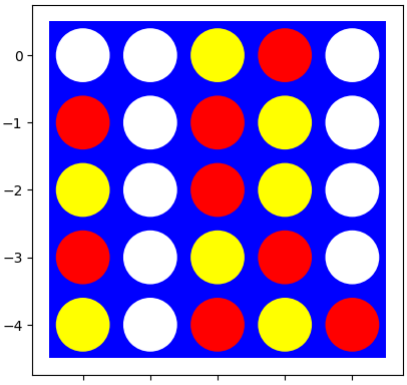

0
−1
−2
−3
−4
0
1
2
3
4

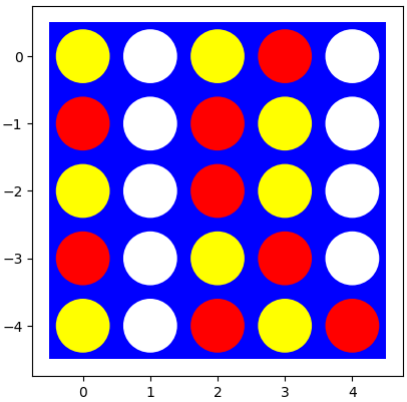

0
−1
−2
−3
−4
0
1
2
3
4

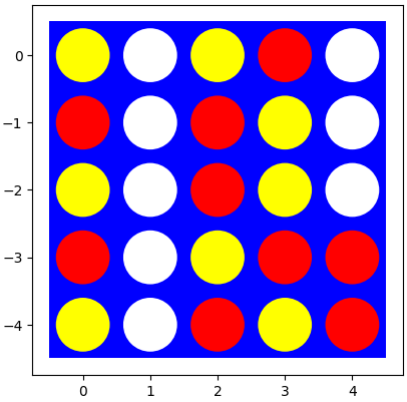

0
−1
−2
−3
−4
0
1
2
3
4

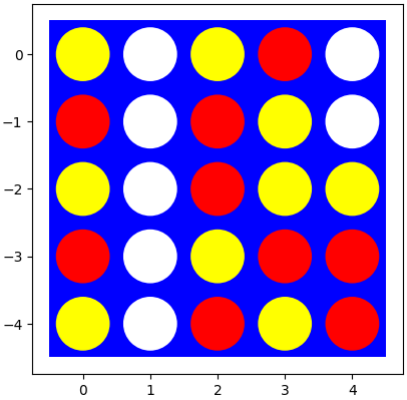

0
−1
−2
−3
−4
0
1
2
3
4

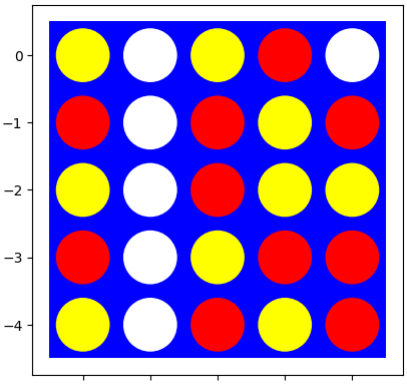

0
−1
−2
−3
−4
0
1
2
3
4

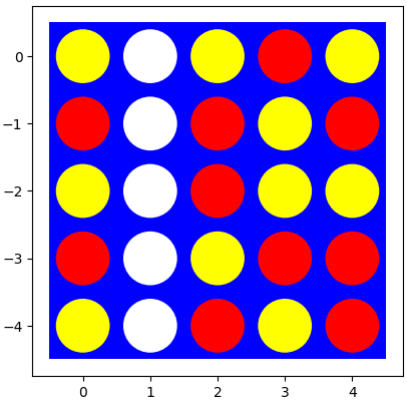

0
−1
−2
−3
−4
0
1
2
3
4

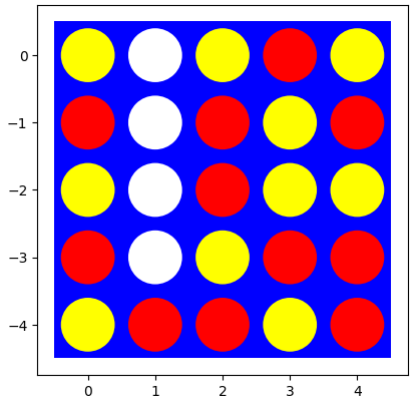

0
−1
−2
−3
−4
0
1
2
3
4

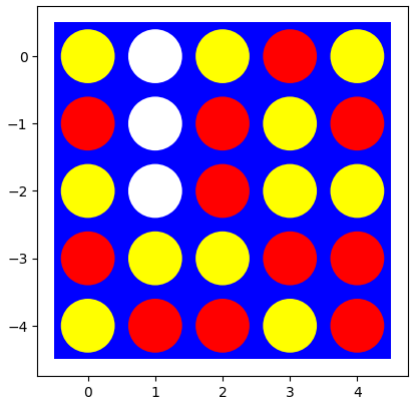

0
−1
−2
−3
−4
0
1
2
3
4

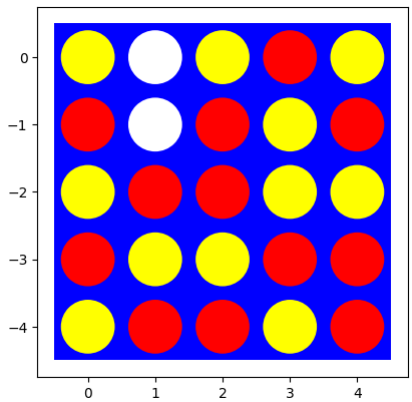

0
−1
−2
−3
−4
0
1
2
3
4


0
−1
−2
−3
−4
0
1
2
3
4


0
−1
−2
−3
−4
0
1
2
3
4

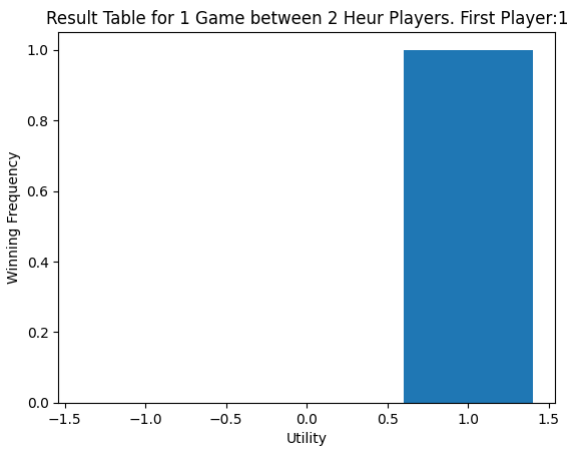






```
winner is player: 1
    1  -1   0
0   1   0   0
```

Out[60]: Text(0.5, 1.0, 'Result Table for 1 Game between 2 Heur Players. First Player:1')

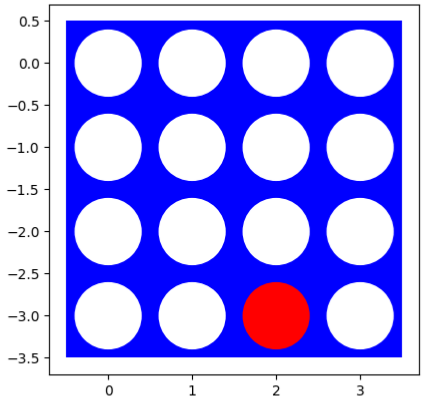


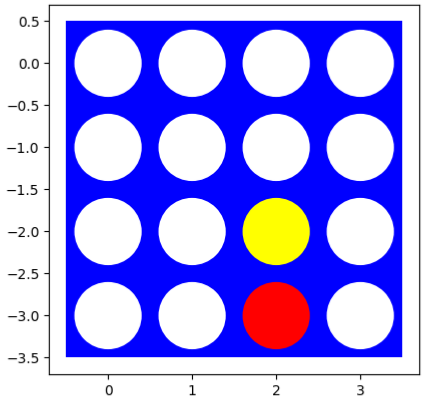
Heuristic with a slightly better heuristic function and higher depth won the game.

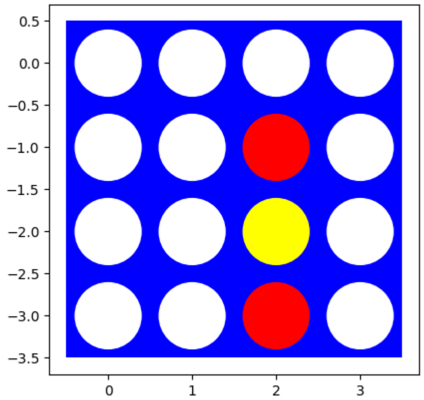
In [155…

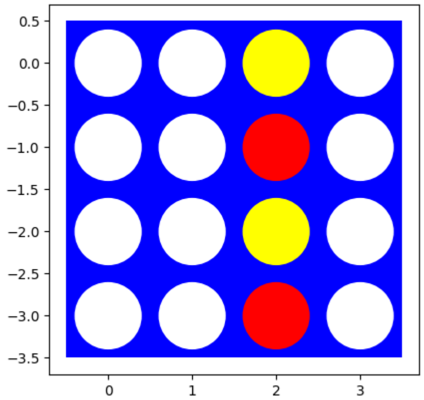
```
first_player=1#First Player 1
result_heur_vs_heur=pd.DataFrame([game_heur(random_player, first_player, True, n=1, h=4, w=4)])
print(result_heur_vs_heur)

result_arr = result_heur_vs_heur.to_numpy()
utility_arr=([1, -1, 0])
```

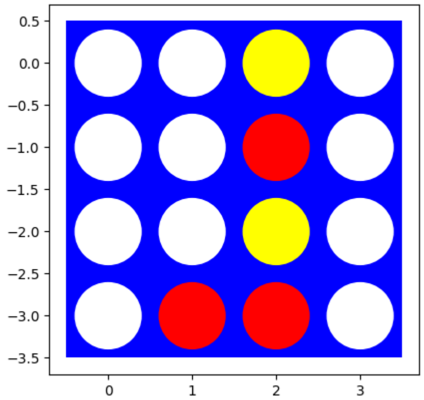
```
plt.bar(utility_arr,result_arr[0])
plt.xlabel("Utility")
plt.ylabel("Winning Frequency")
plt.title("Result Table for 1 Game between 2 Heur Players. First Player:"+str(first_player))
```

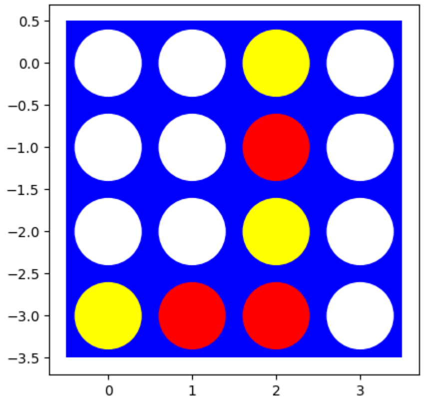


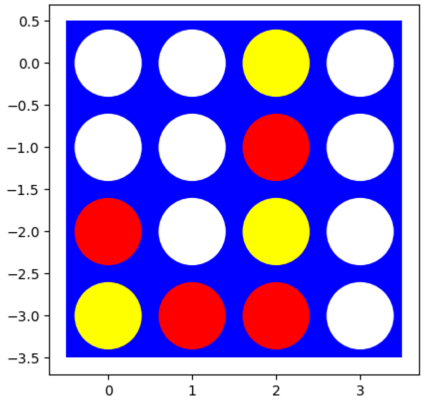


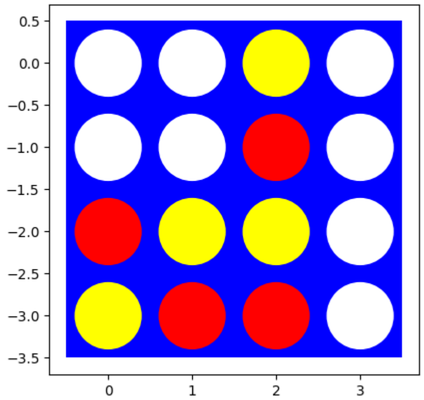


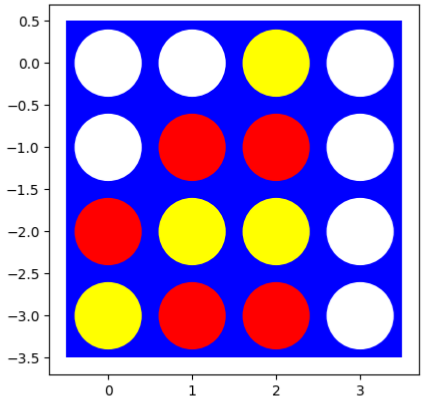


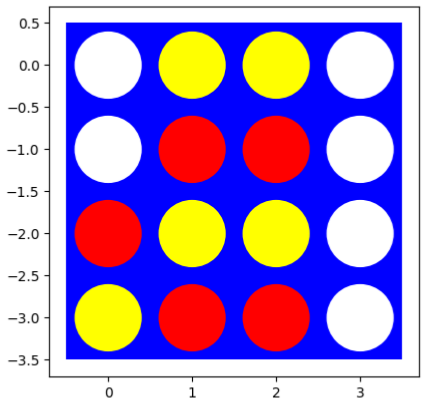

0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0
1
2
3

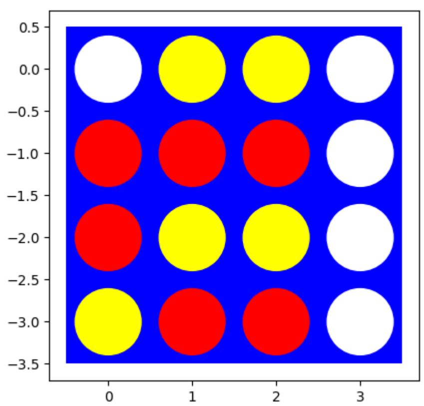

0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0
1
2
3

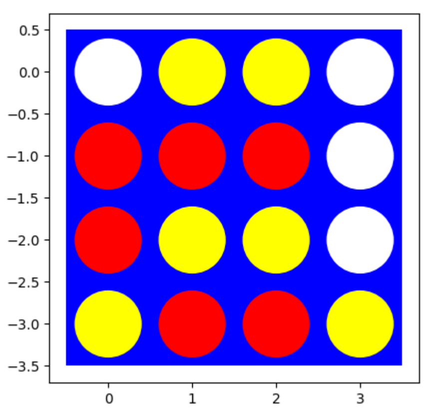

0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0
1
2
3

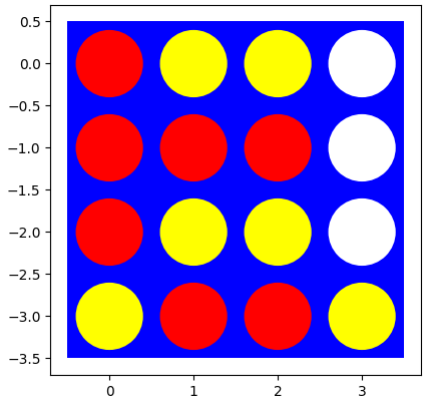

0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0
1
2
3

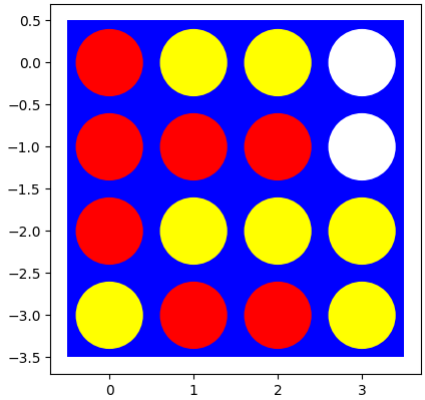

0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0
1
2
3

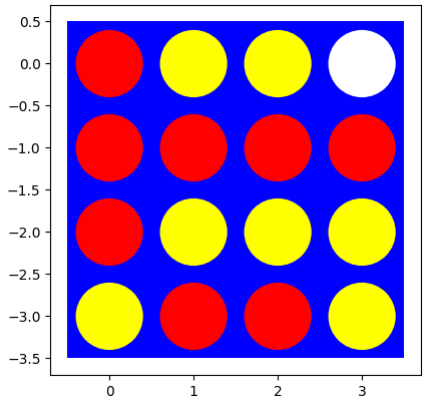

0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0
1
2
3


0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0
1
2
3


0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0
1
2
3

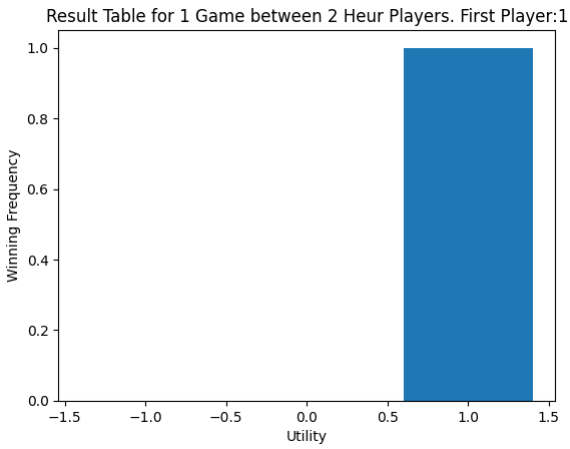






```
winner is player: 1
    1  -1   0
0   1   0   0
```

Out[155]: Text(0.5, 1.0, 'Result Table for 1 Game between 2 Heur Players. First Player:1')

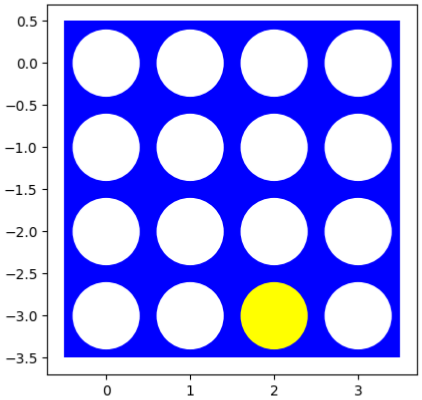


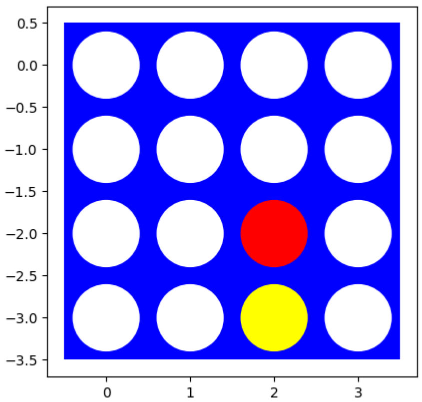
For the smaller board (4x4), the heuristic minimax with a higher depth won the game.

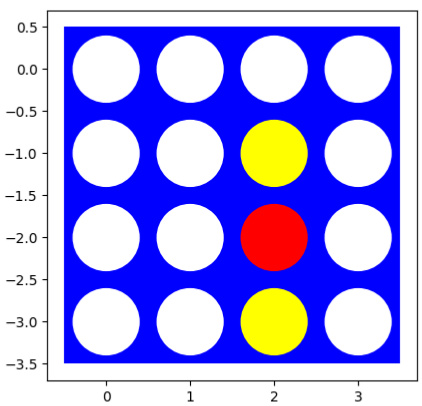
Let's change the first player and begin with the yellow disk (lower search depth).

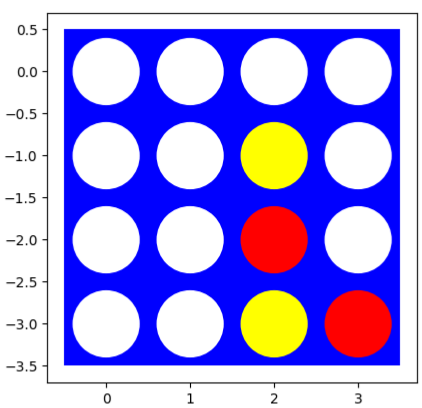
In [58]:
```
first_player=-1#First Player 1
result_heur_vs_heur=pd.DataFrame([game_heur(random_player, first_player, True, n=1, h=4, w=4)])
print(result_heur_vs_heur)
```

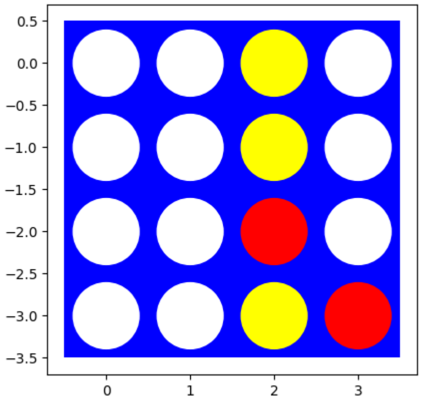
```
result_arr = result_heur_vs_heur.to_numpy()
utility_arr=([1, -1, 0])
plt.bar(utility_arr,result_arr[0])
plt.xlabel("Utility")
plt.ylabel("Winning Frequency")
plt.title("Result Table for 1 Game between 2 Heur Players. First Player:"+str(first_player))
```

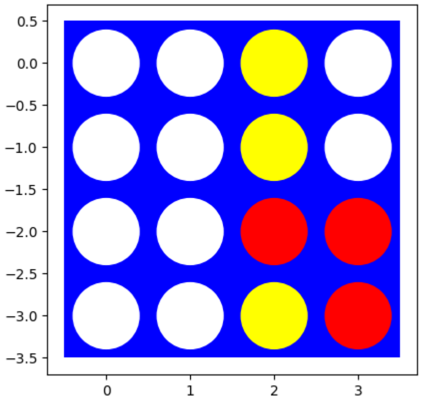


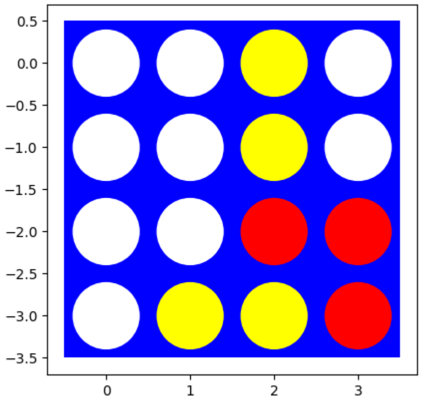


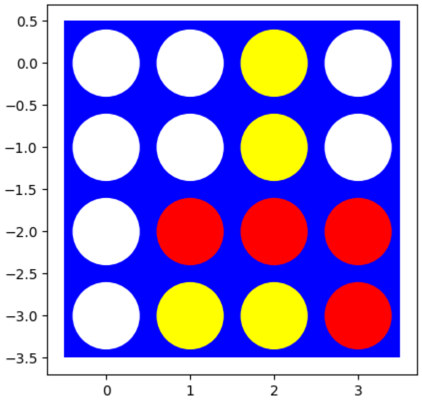


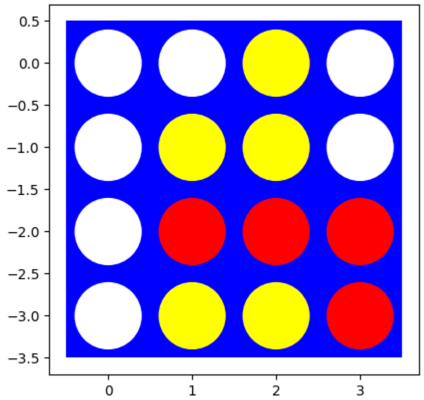


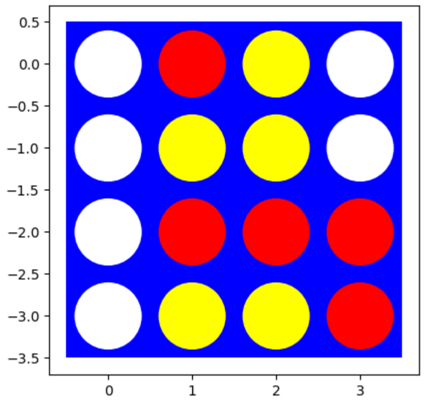

0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0
1
2
3

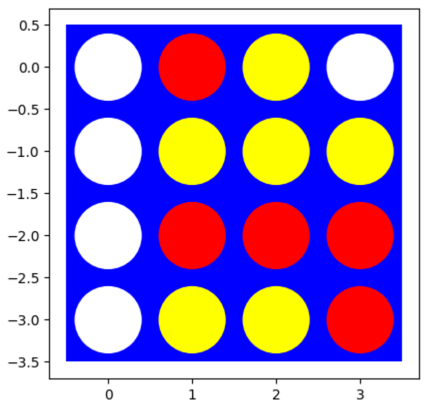

0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0
1
2
3

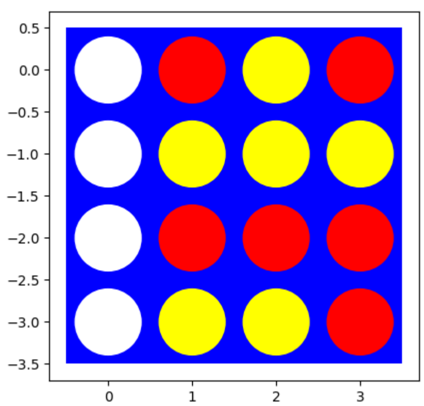

0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0
1
2
3

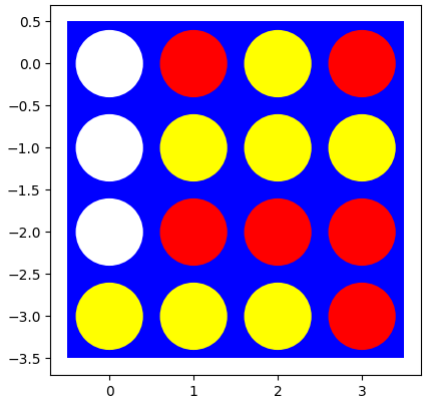

0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0
1
2
3

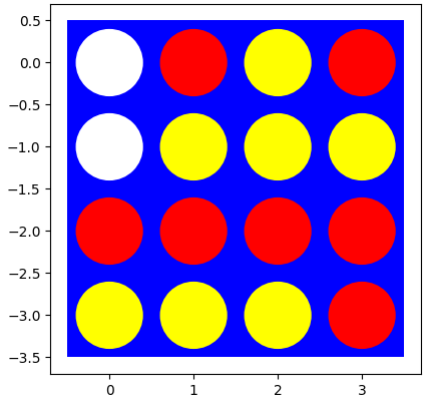

0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0
1
2
3


0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0
1
2
3


0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0
1
2
3

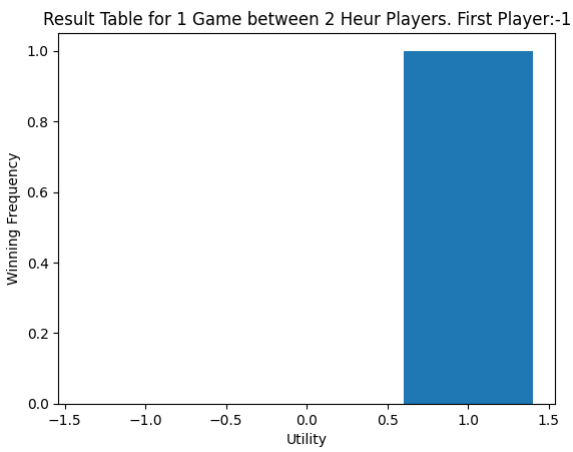

0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
0
1
2
3





```
winner is player: 1
    1  -1   0
0   1   0   0
```

Out[58]: Text(0.5, 1.0, 'Result Table for 1 Game between 2 Heur Players. First Player:-1')



Even though yellow disk started (minimax with lower depth), red won the game with his higher depth and better heuristic values.

Challenge task [+ 10 bonus point will be awarded separately]

Find another student and let your best agent play against the other student's best player. We will set up a class tournament on Canvas. This tournament will continue after the submission deadline.

I will find someone to play!

Graduate student advanced task: Pure Monte Carlo Search and Best First Move [10 point]

Undergraduate students: This is a bonus task you can attempt if you like [+10 bonus point].

Pure Monte Carlo Search

Implement Pure Monte Carlo Search and investigate how this search performs on the test boards that you have used above.

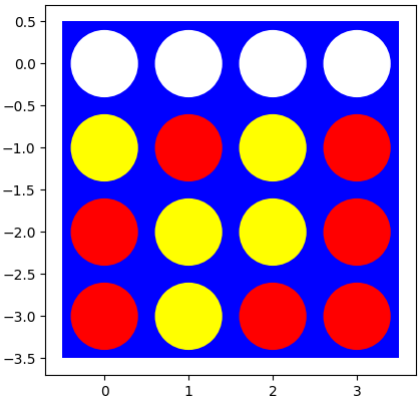
I used the tic-tac-toe example and changed it accordingly for our case.

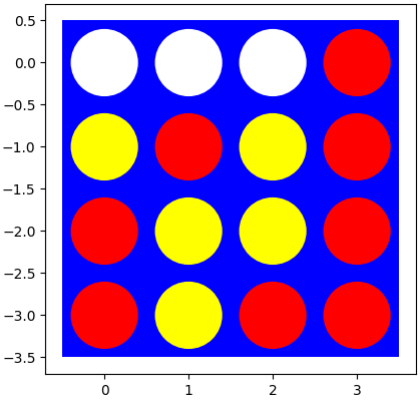
In [222…

```
def playout(state, action, player_id):

    state = action_result(state, action, player_id)
    player_id = switch(player_id)

    while(True):
        # terminal check
        winner = terminal(state)
        if winner is not None:
            return(winner)
```

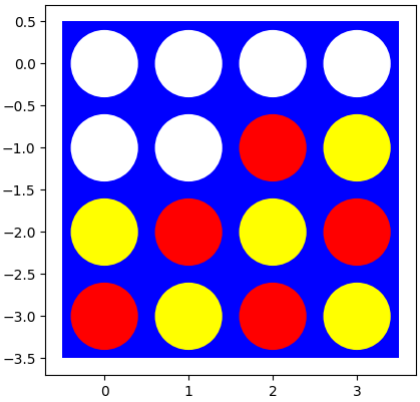
```
        action = random_player(state)
        state = action_result(state, action, player_id)

        # switch player
        player_id = switch(player_id)

def playouts(state, actions, playerid, N):
    return [playout(state, actions, playerid) for i in range(N)]

def pmcs(state, N, player_id):
    acts = actions(state)
    n = math.floor(N/len(acts))

    max = np.mean(playouts(state, acts[0], player_id, N))
    best = acts[0]
    for j in acts:
        current = np.mean(playouts(state, j, player_id, N))
        if current > max:
            max = current
            best = j
    return best
```

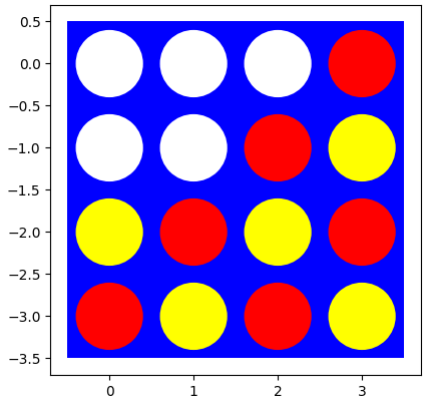
In [212…
```
board = np.array([[0, 0, 0, 0],
                  [-1, 1, -1, 1],
                  [1, -1, -1, 1],
                  [1, -1, 1, 1]])
visualize(board)
visualize(action_result(board, pmcs(board, 1000, 1),1))
```



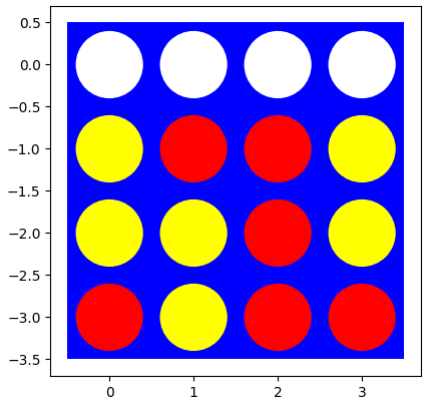


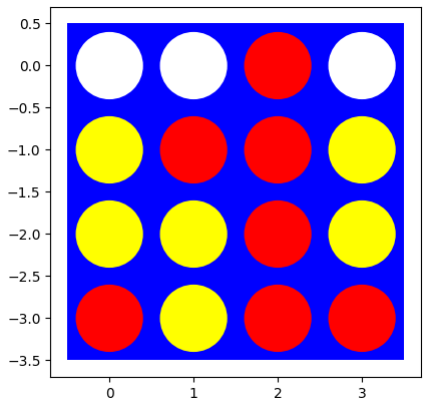
In [158…
```
board = np.array([[0, 0, 0, 0],
                  [0, 0, 1, -1],
                  [-1, 1, -1, 1],
                  [1, -1, 1, -1]])
visualize(board)
visualize(action_result(board, pmcs(board, 1000, 1),1))
```



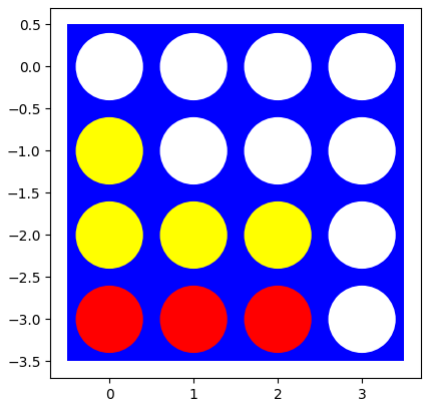


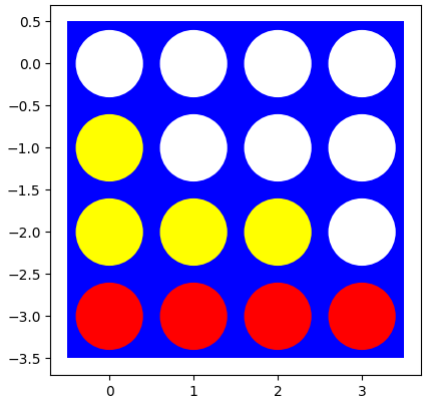
In [159…
```
board = np.array([[0, 0, 0, 0],
                  [-1, 1, 1, -1],
                  [-1, -1, 1, -1],
                  [1, -1, 1, 1]])

visualize(board)
visualize(action_result(board, pmcs(board, 1000, 1),1))
```



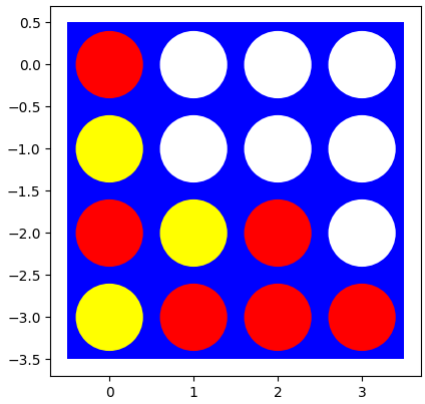


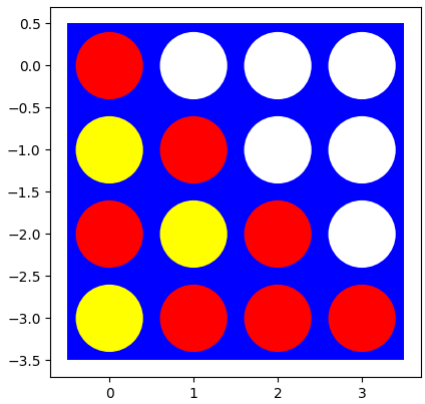
In [160…
```
board = np.array([[0, 0, 0, 0],
                  [-1, 0, 0, 0],
                  [-1, -1, -1, 0],
                  [1, 1, 1, 0]])
visualize(board)
visualize(action_result(board, pmcs(board, 1000, 1),1))
```



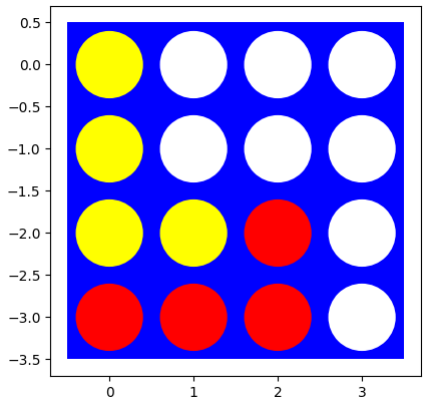


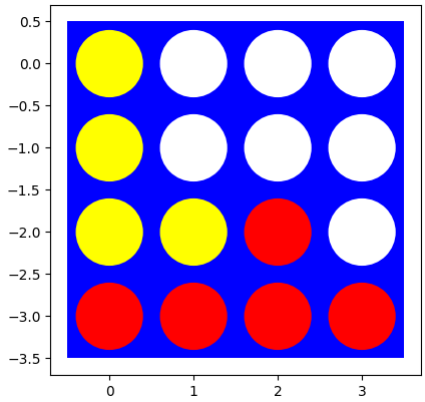
In [161…
```
board = np.array([[1, 0, 0, 0],
                  [-1, 0, 0, 0],
                  [ 1, -1, 1, 0],
                  [-1, 1, 1, 1]])

visualize(board)
visualize(action_result(board, pmcs(board, 1000, 1),1))
```





In [162…
```
board = np.array([[-1, 0, 0, 0],
                  [-1, 0, 0, 0],
                  [-1, -1, 1, 0],
                  [1, 1, 1, 0]])

visualize(board)
visualize(action_result(board, pmcs(board, 1000, 1),1))
```





Monte-Carlo could find the terminal and the shortest path for all. Minimax and Heuristic-Minimax found the terminal in the second move for this board.

Best First Move

Use Oure Monte Carlo Search to determine what the best first move is? Describe under what assumptions this is the "best" first move.

I used 3000 iterations for empty board (6x7).

In [163…
```
board=empty_board()
pmcs(board, 3000, 1)
```

Out[163]: 3

As I explained in "the first few moves" part, selecting the middle column returns a better winning performance.