

# Adversarial Search: Playing Connect 4

Student Name: [Add your name]

## 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 \(https://en.wikipedia.org/wiki/Connect\\_Four\)](https://en.wikipedia.org/wiki/Connect_Four))

Note that [Connect-4 has been solved \(https://en.wikipedia.org/wiki/Connect\\_Four#Mathematical\\_solution\)](https://en.wikipedia.org/wiki/Connect_Four#Mathematical_solution) 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/> (<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)

Initial State: The initial state is the empty board of 6 rows and 7 columns traditionally.

Actions: The actions are the possible moves that can be made by the player. In this case, the possible moves are the columns that are not full. The "open" column.

**Transition Model:** The transition model is the result of putting a piece in an available column. The piece will then fall to the lowest available row in that column.

**Goal State:** The goal state is when one player played 4 pieces of the same color in a row. This row can be horizontal, vertical, or diagonal.

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

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

I believe that a normal agent at a given time will have to have a state size of  $3^{42}$ . This is because there are 3 possible states (empty, red, yellow) for each of the 42 spaces on the board. The state size will be less than  $3^{42}$  because it does not count for impossible board states. A piece cannot be floating in midair and will always be at the bottommost row. The number of red and yellow pieces will always be equal so that will also decrease the amount of total board states there are.

The kicker is, that this is a depth first search miniMax, so it will have to go through all of the possible moves for each of the 42 spaces on the board. This is a very large number, and will take a very long time to compute. at the very least it will take  $3^{42}!$  states to look at every possible state.

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

Use a numpy character array as the board.

```
In [434]: import numpy as np
import random
import pandas as pd

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  $6 \times 7$  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 form: `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 [390]: `# Visualization code by Randolph Rankin`

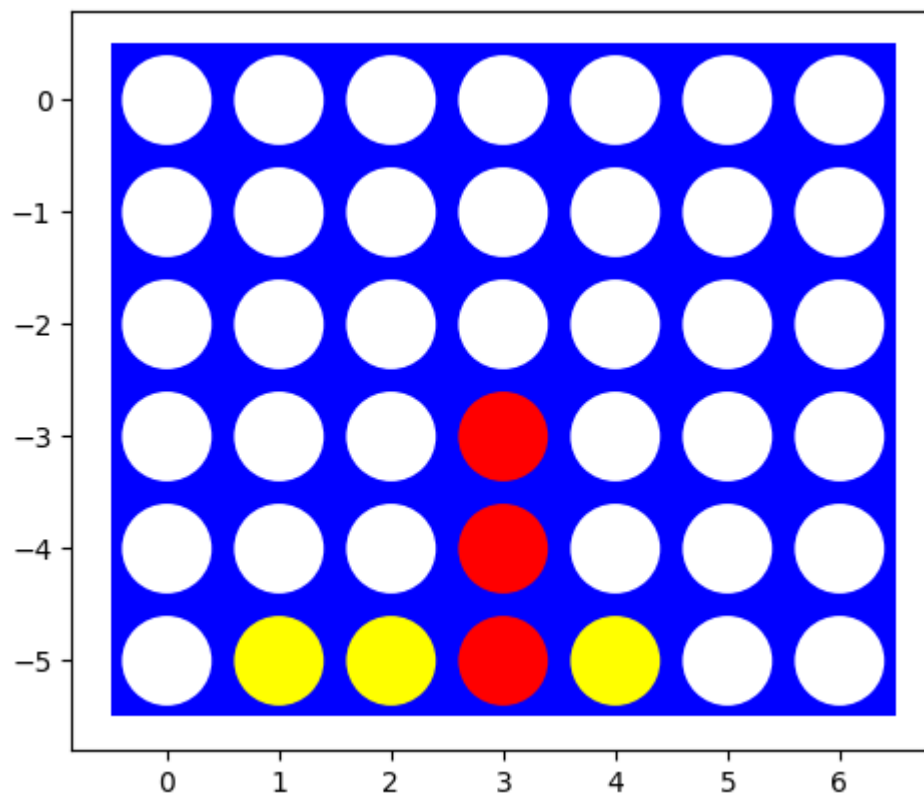
```
import matplotlib.pyplot as plt

def visualize(board):
    plt.axes()
    rectangle=plt.Rectangle((-0.5,len(board)*-1+0.5),len(board[0]),len(board))
    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()

exBoard = [[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(exBoard)
```



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).

```

In [391]: # Your code/ answer goes here.

# actions(s) returns a list of all moves the player can take
# result(s,a) returns the resultant board after the player chooses a move from
# terminal(s) returns true if there is a connect 4 or if the board is full
# utility(s) returns who wins the game

player1 = 1
player2 = -1
tie = 0

board = empty_board(shape = (6,7))

#we need a function to check is a move is valid or not.
def isBoardFull(board):
    for i in range(len(board[0])):
        if board[0][i] == 0:
            return False
    return True

#this function provides all available columns for the player to drop their pi
#function to check if the move is valid or not
def validMove(board, col):
    if board[0][col] == 0:
        return True
    else:
        return False

#function to determine who's turn it is
#referenced from: https://numpy.org/doc/stable/reference/generated/numpy.count_nonzero
# def turn(board):
#     if np.count_nonzero(board == 1) > np.count_nonzero(board == -1):
#         return player2
#     else:
#         return player1

#function to determine who's turn it is
def turn(board):
    p1count = 0
    p2count = 0
    for i in range(len(board)):
        for j in range(len(board[0])):
            if board[i][j] == 1:
                p1count+=1
            elif board[i][j] == -1:
                p2count+=1
    if p1count > p2count:
        return player2
    else:
        return player1

#function to determine who's

def availableMoves(board):
    availableMoves = []

```

```

for i in range(len(board[0])):
    if validMove(board, i):
        availableMoves.append(i)
return availableMoves

def result(board, player, action):
    if validMove(board, action):
        newBoard = np.copy(board)
        for i in range(len(board)-1, -1, -1):
            if newBoard[i][action] == 0:
                if player == 1:
                    newBoard[i][action] = 1
                if player == -1:
                    newBoard[i][action] = -1

                break
        return newBoard
    else:
        return "Invalid move"

#function to determine if the game is over by checking if there is a connect
def terminal(board):
    #check for horizontal connect 4
    for i in range(len(board)):
        for j in range(len(board[0])-3):
            if board[i][j] == board[i][j+1] == board[i][j+2] == board[i][j+3]:
                return board[i][j]
    #check for vertical connect 4
    for i in range(len(board)-3):
        for j in range(len(board[0])):
            if board[i][j] == board[i+1][j] == board[i+2][j] == board[i+3][j]:
                return board[i][j]
    #check for diagonal connect 4
    for i in range(len(board)-3):
        for j in range(len(board[0])-3):
            if board[i][j] == board[i+1][j+1] == board[i+2][j+2] == board[i+3][j+3]:
                return board[i][j]
    for i in range(len(board)-3):
        for j in range(len(board[0])-3):
            if board[i][j+3] == board[i+1][j+2] == board[i+2][j+1] == board[i+3][j]:
                return board[i][j+3]
    #check if the board is full
    if isBoardFull(board):
        return tie
    #if the game is not over
    return None

#this function determines the utility of the board
def utility(board):
    if(terminal(board) == player1):
        visualize(board)
        return print("Player 1 wins!")
    elif(terminal(board) == player2):
        visualize(board)
        return print("Player 2 wins!")
    elif(terminal(board) == tie):

```

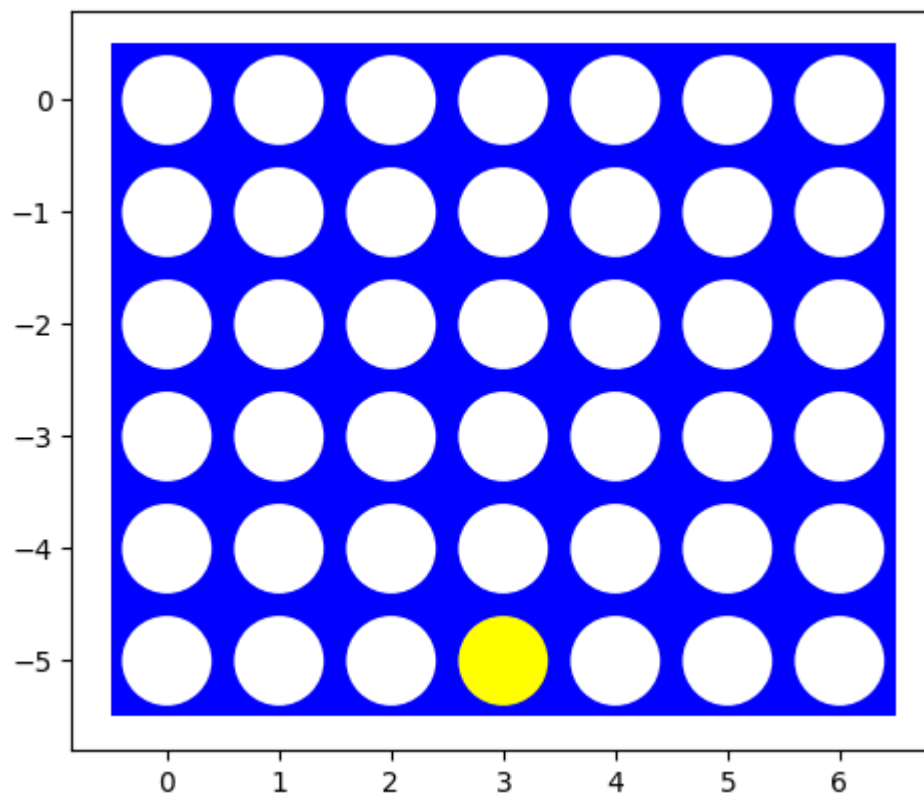
```

visualize(board)
return print("The game is a tie!")

def otherPlayer(player):
    if player == 1:
        return -1
    else:
        return 1

```


In [392]: `visualize(result(board, player2, 3))`



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 or -1 for player red and yellow, respectively. See [Experiments section for tic-tac-toe \(https://nbviewer.org/github/mhahsler/CS7320-AI/blob/master/Games/tictactoe\\_and\\_or\\_tree\\_search.ipynb#Experiments\)](https://nbviewer.org/github/mhahsler/CS7320-AI/blob/master/Games/tictactoe_and_or_tree_search.ipynb#Experiments) for an example.

```
In [393]:  # Your code/ answer goes here.  
  
def randomPlayer(board, player):  
    return np.random.choice(availableMoves(board))
```

Let two random agents play against each other 1000 times. Look at the [Experiments section for tic-tac-toe \(https://nbviewer.org/github/mhahsler/CS7320-AI/blob/master/Games/tictactoe\\_and\\_or\\_tree\\_search.ipynb#Experiments\)](https://nbviewer.org/github/mhahsler/CS7320-AI/blob/master/Games/tictactoe_and_or_tree_search.ipynb#Experiments) 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 [394]: # Your code/ answer goes here.
def stimulateGame(agent1, agent2, numGames=1000, tester=False):
    player1Wins = 0
    player2Wins = 0
    ties = 0
    for i in range(numGames):
        board = empty_board()
        player = player1
        agent = agent1
        while terminal(board) == None:
            if player == player1:
                action = agent(board, player)
                board = result(board, player, action)
                player = player2
                if tester:
                    visualize(board)
            if player == player2:
                agent = agent2
                action = agent(board, player)
                board = result(board, player, action)
                player = player1
                if tester:
                    visualize(board)
            if terminal(board) == player1:
                player1Wins += 1
            elif terminal(board) == player2:
                player2Wins += 1
            else:
                ties += 1
    results = {"agent1 wins": player1Wins, "agent2 wins": player2Wins, "number of ties": ties}
    if tester:
        visualize(board)
    return results
# print (randomPlayer(board))
print("Random vs Random agents")
print(stimulateGame(randomPlayer, randomPlayer, 1000, False))
```

Random vs Random agents

{'agent1 wins': 502, 'agent2 wins': 495, 'number of ties': 3}

## 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 \(https://nbviewer.org/github/mhahsler/CS7320-AI/blob/master/Games/tictactoe\\_alpha\\_beta\\_tree\\_search.ipynb\)](https://nbviewer.org/github/mhahsler/CS7320-AI/blob/master/Games/tictactoe_alpha_beta_tree_search.ipynb).

#### 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](https://nbviewer.org/github/mhahsler/CS7320-AI/blob/master/Games/tictactoe_alpha_beta_tree_search.ipynb).

[AI/blob/master/HOWTOs/store\\_agent\\_state\\_information.ipynb](#)) This is essential to be able play against agents from other students later.

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

```

In [514]: # Your code/ answer goes here.
#https://medium.com/analytics-vidhya/artificial-intelligence-at-play-connect-
#https://www.pythonpool.com/numpy-infinity/#:~:text=return%20num,Import%20a%2

# def getValidLocations(board):
#     validLocations = []
#     for col in range(len(board[0])):
#         if validMove(board, col):
#             validLocations.append(col)
#     return validLocations

# def miniMaxAlphaBeta (board, difficulty, player, alpha , beta, maximizingPlayer):
#     #isGameOver = terminal(board)
#     validMoves = availableMoves(board)
#     if terminal(board) != None:
#         return terminal(board)
#     if maximizingPlayer:
#         value = -np.inf
#         moveChoice = random.choice(validMoves)
#         for action in validMoves:
#             score = miniMaxAlphaBeta(result(board, player, action), difficulty, player, alpha, beta, False)
#             if score > value:
#                 value = score
#                 moveChoice = action
#             alpha = max(alpha, value)
#             if alpha >= beta:
#                 break
#         return moveChoice, value
#     else:
#         value = np.inf
#         for action in validMoves:
#             score = miniMaxAlphaBeta(result(board, player, action), difficulty, player, alpha, beta, True)
#             if score < value:
#                 value = score
#                 moveChoice = action
#             beta = min(beta, value)
#             if alpha >= beta:
#                 break
#         return moveChoice, value

def miniMaxAgent(board, player):
    return miniMaxSearch(board, player)["move"]

def miniMaxSearch(board, player):
    value, moveChoice = miniMaxMax(board, player, 0, -np.inf, np.inf)
    return {"move": moveChoice, "value": value}

def miniMaxMax(board, player, depth, alpha, beta):
    validMoves = availableMoves(board)
    validMoves.sort(key=lambda x: abs(x-len(board[0]))//2)
    if terminal(board) != None:
        return terminal(board), None
    # if depth == 3:
    #     return miniMaxSearch(board), None

```

```

value, moveChoice = -np.inf, None
for action in validMoves:
    value2, action2 = miniMaxMin(result(board, player, action), player, d
    if value2 > value:
        value, moveChoice = value2, action
        alpha = max(alpha, value)

    if value >= beta:
        return value, moveChoice
return value, moveChoice

def miniMaxMin(board, player, depth, alpha, beta):
    validMoves = availableMoves(board)
    validMoves.sort(key=lambda x: abs(x-len(board[0])//2))
    if terminal(board) != None:
        return terminal(board), None
    # if depth == 3:
    #     return miniMaxSearch(board), None

    value, moveChoice = np.inf, None
    for action in validMoves:
        value2, action2 = miniMaxMax(result(board, otherPlayer(player), actio
        if value2 < value:
            value, moveChoice = value2, action
            beta = min(beta, value)
        if value <= alpha:
            return value, moveChoice
    return value, moveChoice

def miniMaxVminiMax(board, agent1):
    agent = agent1
    while terminal(board) == None:
        action = miniMaxAgent(board, agent)
        board = result(board, agent, action)
        visualize(board)
        agent = otherPlayer(agent)
    return terminal(board)

def stimulation(board, agent1, agent2):
    agent = agent1
    player = turn(board)
    while terminal(board) == None:
        action = agent(board, player)
        board = result(board, agent, action)
        visualize(board)
        if agent == agent1:
            agent = agent2
            player = otherPlayer(player)
        else:
            agent = agent1
            player = turn(board)
    return terminal(board)

def play(board, agent1, agent2, N = 100, tester = False):
    player1wins = 0

```

```

player2wins = 0
ties = 0
if N == 1:
    board = np.copy(board)
    player, fun = turn(board), agent1
    while terminal(board) == None:
        action = fun(board, player)
        board = result(board, player, action)
        if tester:
            visualize(board)
        player, fun = otherPlayer(player), agent2 if fun == agent1 else a
    return terminal(board)

else:
    for i in range(N):
        board = empty_board()
        player, fun = turn(board), agent1
        while terminal(board) == None:
            a = fun(board, player)
            board = result(board, player, a)
            if tester:
                visualize(board)
            player, fun = otherPlayer(player), agent2 if fun == agent1 el

        if terminal(board) == 1:
            player1wins += 1
            # visualize(board)
        elif terminal(board) == -1:
            player2wins += 1
        else:
            ties += 1

    results = {"player1 wins": player1wins, "player2 wins": player2wi
return results

```

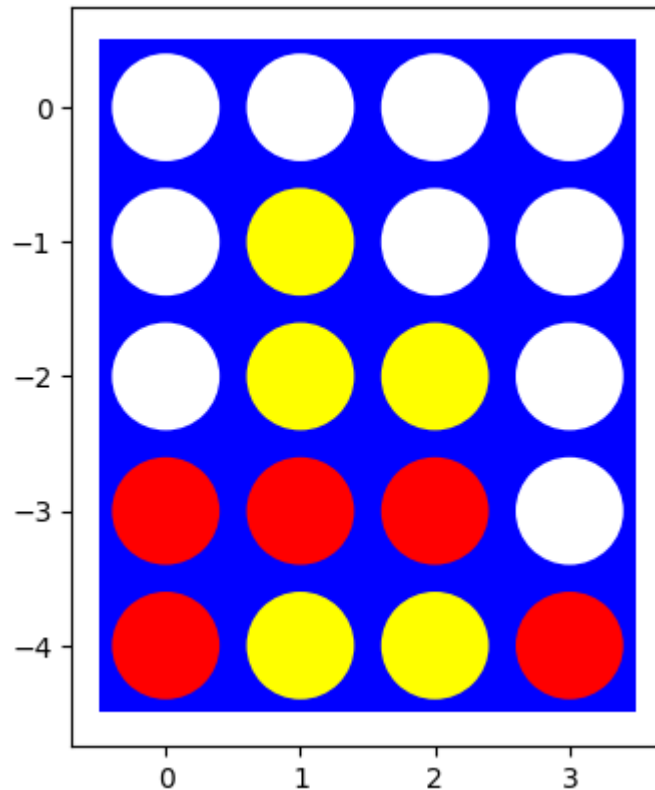
Experiment with some manually created boards (at least 5) to check if the agent spots winning opportunities.

## Test 1:

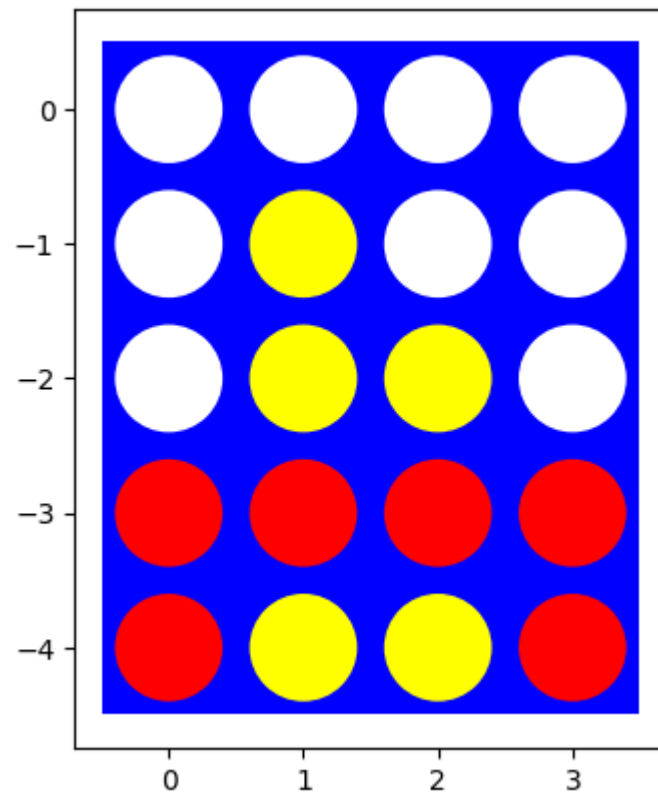
```
In [426]: exampleBoard1 = [[0, 0, 0, 0],
                             [ 0, -1, 0, 0],
                             [0, -1, -1, 0],
                             [1, 1, 1, 0],
                             [1, -1, -1, 1]]

visualize(exampleBoard1)
print ("It is player:" , turn(exampleBoard1), "'s turn")

print(miniMaxAgent(exampleBoard1, turn(exampleBoard1)))
#miniMaxVminiMax(exampleBoard1, turn(exampleBoard1))
#stimulation(exampleBoard1, miniMaxAgent, miniMaxAgent)
play(exampleBoard1, miniMaxAgent, miniMaxAgent, 1)
```



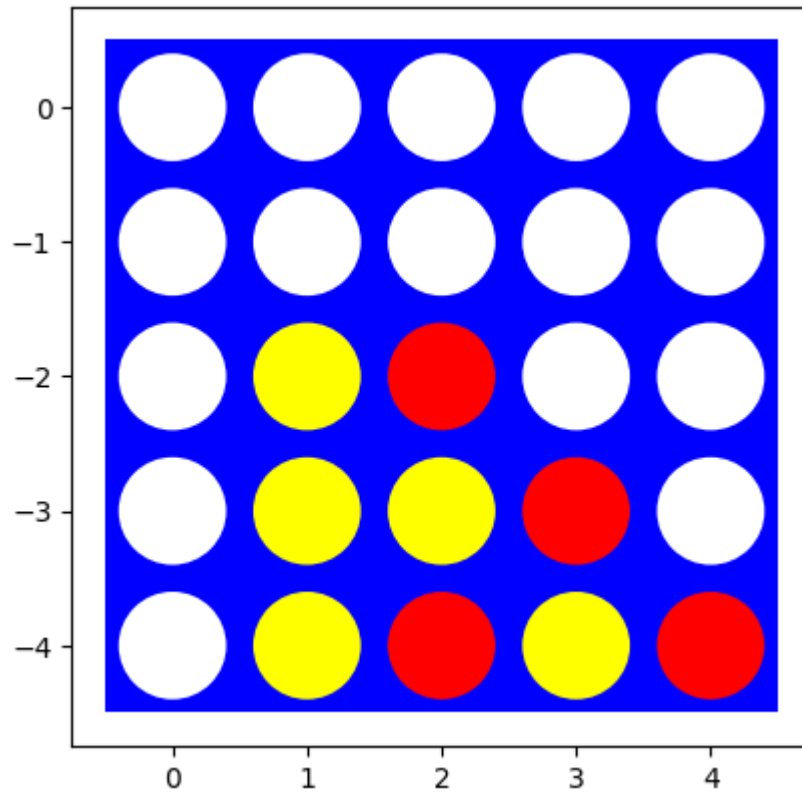
It is player: 1 's turn  
3



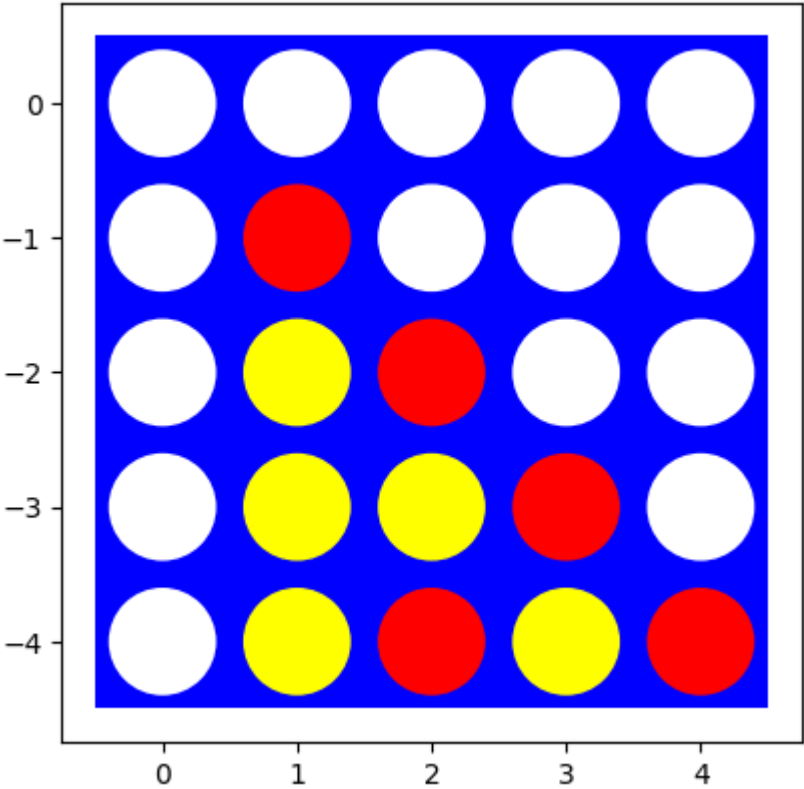
Out[426]: 1

**Test 2**

```
In [442]: exampleBoard2 = [[0, 0, 0, 0,0],
                             [ 0, 0, 0, 0,0],
                             [0, -1, 1, 0,0],
                             [0, -1, -1, 1,0],
                             [0, -1, 1, -1,1]]
visualize(exampleBoard2)
# print ("current player:", turn(exampleBoard2))
# print(miniMaxAgent(exampleBoard2, turn(exampleBoard2)))
play(exampleBoard2, miniMaxAgent, miniMaxAgent, 1)
```







Out[442]: 1

Test 3

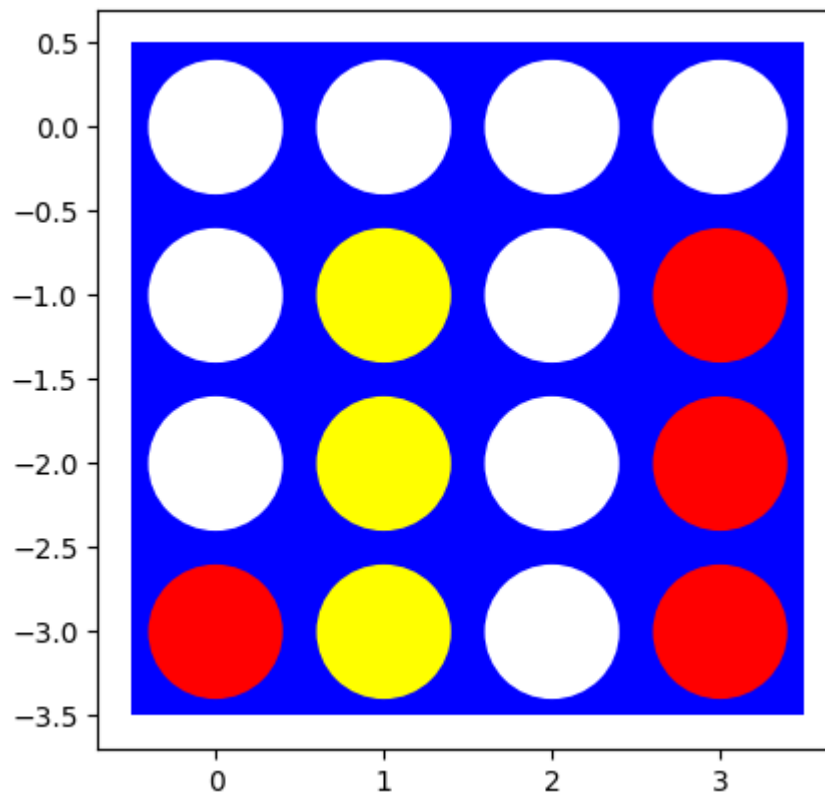
```

In [428]: exampleBoard3 = [
            [ 0, 0, 0, 0],
            [0, -1, 0, 1],
            [0, -1, 0, 1],
            [1, -1, 0, 1]]

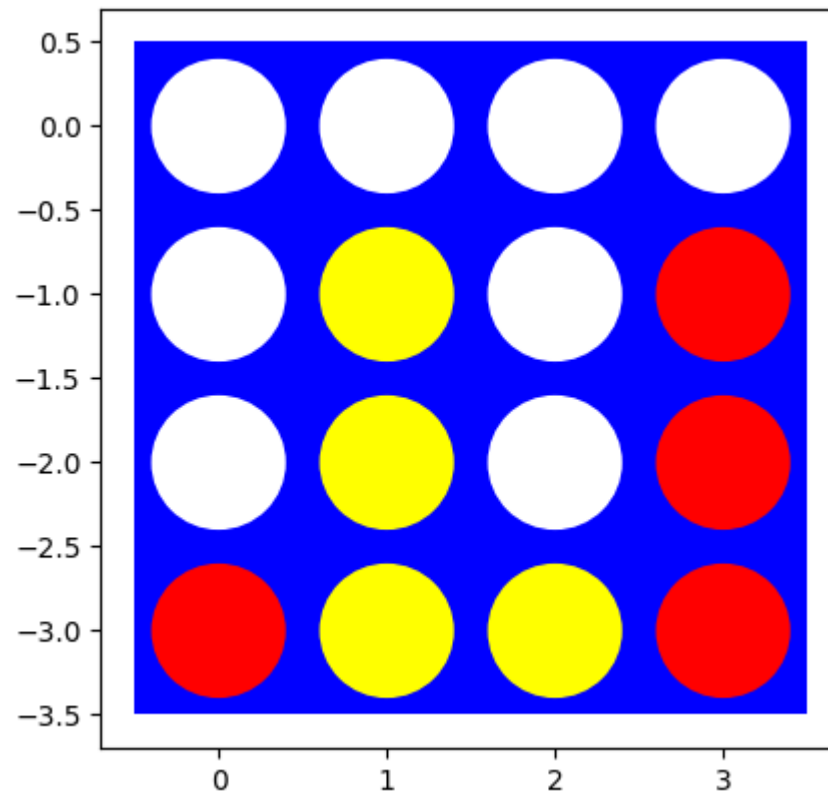
visualize(exampleBoard3)
print ("current player:", turn(exampleBoard3))
#print (miniMaxSearch(exampleBoard3, turn(exampleBoard3)))
print (miniMaxSearch(exampleBoard3, -1))
#visualize(exampleBoard3)

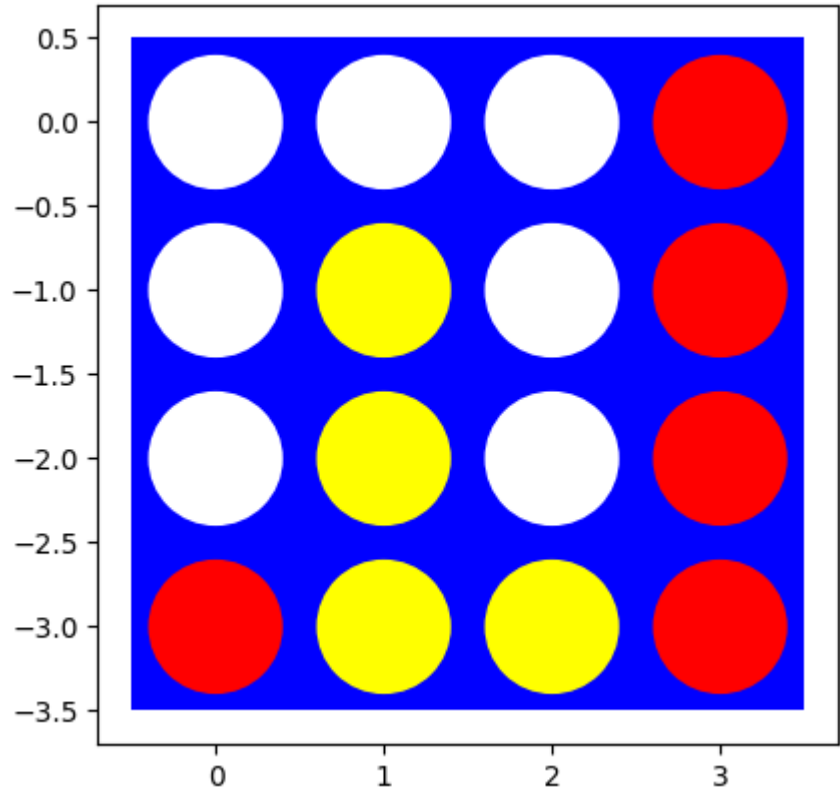
#stimulateGame(miniMaxAgent(exampleBoard3,player1), miniMaxAgent(exampleBoard3,player2))
#miniMaxVminiMax(exampleBoard3, turn(exampleBoard3))
play(exampleBoard3, miniMaxAgent, miniMaxAgent, 1)

```



```
current player: -1  
{'move': 2, 'value': 0}
```

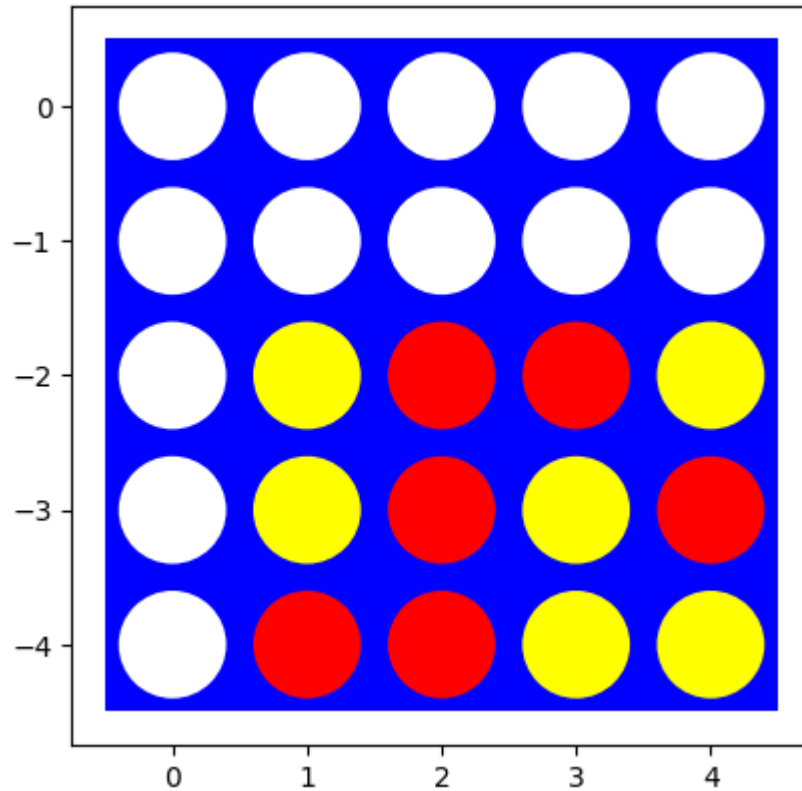


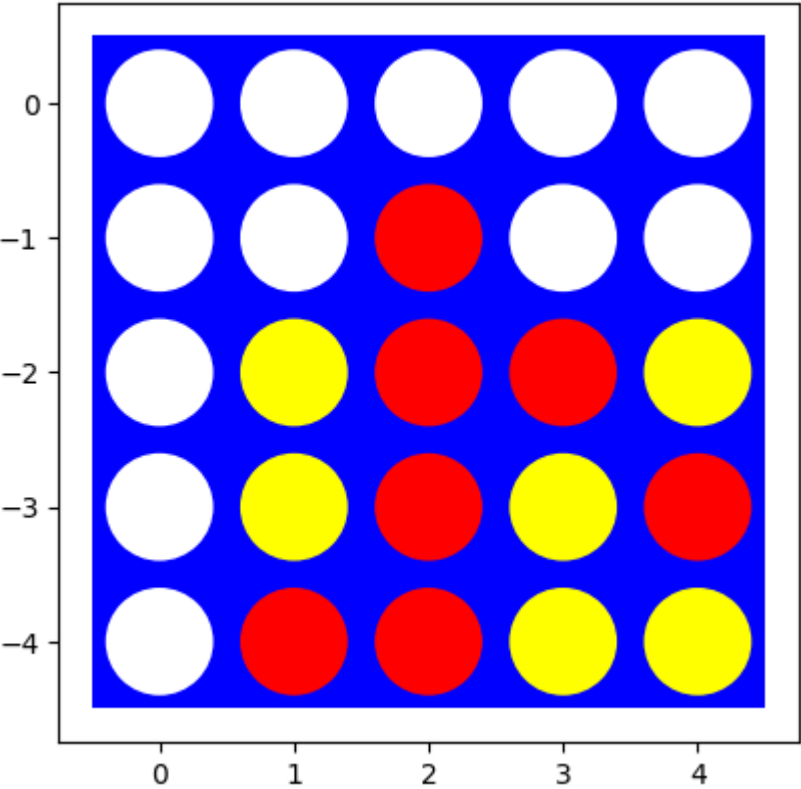


Out[428]: 1

Test 4

```
In [429]: exampleBoard4 = [[0, 0, 0, 0, 0],
                             [ 0, 0, 0, 0, 0],
                             [0, -1, 1, 1, -1],
                             [0, -1, 1, -1, 1],
                             [0, 1, 1, -1, -1]]
visualize(exampleBoard4)
#miniMaxVminiMax(exampleBoard4, turn(exampleBoard4))
play(exampleBoard4, miniMaxAgent, miniMaxAgent, 1)
```

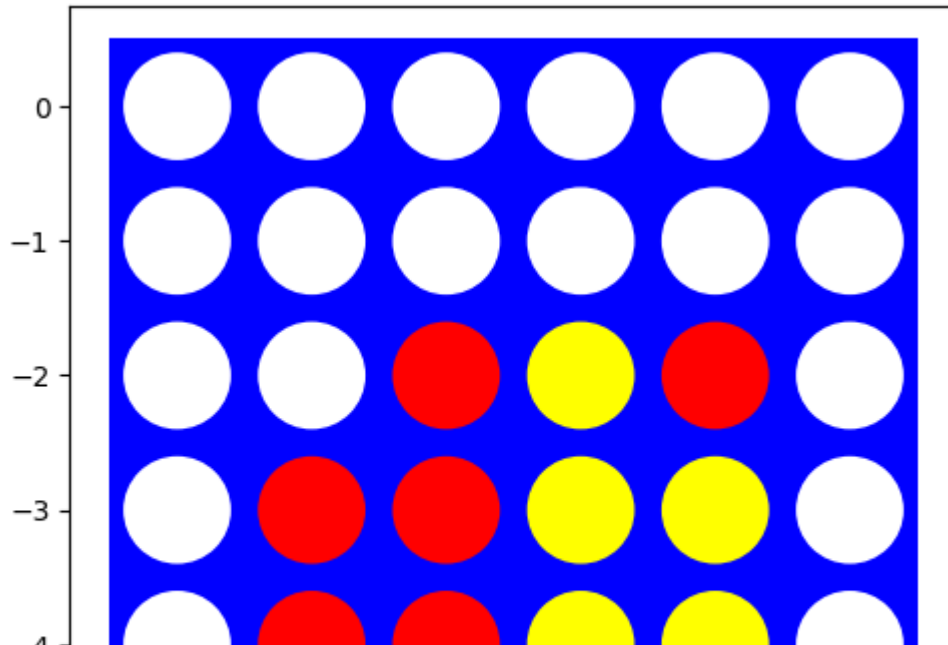




Out[429]: 1

Test 5

```
In [430]: exampleBoard5 = [
    [ 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, -0, 1, -1, 1, 0],
    [0, 1, 1, -1, -1, 0],
    [0, 1, 1, -1, -1, 0]]
visualize(exampleBoard5)
miniMaxVminiMax(exampleBoard5, turn(exampleBoard5))
```



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.

I could not run this code on a 6x7 empty board. I found that increasing the amount of rows would really slow down the code. Also, partially filling in the board would speed up the code tremendously. This is because the code above did not have: a depth limit, a heuristic value, or move ordering. The tests in the below code is significantly faster than the code above. The speed data is below in a chart.

### 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?

```

In [438]: # Example.
#This sort, sorts the list by the absolute value of the difference between th
moveOrderedBoard = empty_board()

actionsOfMoveOrderedBoard = availableMoves(moveOrderedBoard)
print ("before the sort it looks like this: ", actionsOfMoveOrderedBoard)
actionsOfMoveOrderedBoard.sort(key=lambda x: abs(x-len(board[0])/2))
print ("after the sort it looks like this: ", actionsOfMoveOrderedBoard)

#Below is data for the boards that I used in the tests above with and without

dataTable = pd.DataFrame(columns = ['Test Case', 'Time Taken Without move ord
dataTable.loc[0] = ['Test 1', 3.333333, 0.900000]
dataTable.loc[1] = ['Test 2', 2.222200, 0.700000]
dataTable.loc[2] = ['Test 3', 2.111111, 2.300000]
dataTable.loc[3] = ['Test 4', 6.600000, 0.700000]
dataTable.loc[4] = ['Test 5', 20.400000, 7.800000]
dataTable.head()

```

before the sort it looks like this: [0, 1, 2, 3, 4, 5, 6]

after the sort it looks like this: [3, 2, 4, 1, 5, 0, 6]

Out[438]:

	Test Case	Time Taken Without move ordering	Time Taken WITH move ordering
0	Test 1	3.333333	0.9
1	Test 2	2.222200	0.7
2	Test 3	2.111111	2.3
3	Test 4	6.600000	0.7
4	Test 5	20.400000	7.8

## NOTE -----

~~~I have identified a bug with my code that I am working on fixing. Above you should notice that yellow does not go for an obvious win. The code block below is me trying to fix the bug.~~~

RESOLVED



```

In [439]: ▶ def centerPriority(board):
            center = len(board[0])//2
            if validMove(board, center):
                return center
            else:
                return random.choice(availableMoves(board))

            #function that orders a list of moves to start with the center column, then t
def centerOrder(board):
    center = len(board[0])//2
    if validMove(board, center):
        return center
    else:
        moves = availableMoves(board)
        moves.sort(key=lambda x: abs(x-center))
        return moves[0]

def moveOrderedMiniMaxSearch(board, player, tester = False):
    validMoves = availableMoves(board)
    validMoves.sort(key=lambda x: abs(x-len(board[0])//2))
    print ("the sorted available moves are:", validMoves)
    moveValues = []
    for action in validMoves:
        moveValues.append(miniMaxSearch(result(board, player, action), player
    moveValues.sort(key = lambda x: x["value"], reverse = True)
    for i in range(len(moveValues)):
        if tester:
            print ("I'm finna print this: ",moveValues[i]["value"])
            if moveValues[i]["value"] == -1:
                print ("this was the bad column it wanted to go to: ", moveVa
    if moveValues[0]["value"] == 0:
        if tester: print("The center is: ",centerPriority(board))
        center = centerPriority(board)
        if center in validMoves:
            moveValues[0]["move"] = center
            return moveValues[0]
    return moveValues[0]

#print(moveOrderedMiniMaxSearch(exampleBoard3, turn(exampleBoard3)))
def MMinimaxAgent(board, player):
    return moveOrderedMiniMaxSearch(board, player)["move"]

def MMinimaxStim(board, agent1):
    agent = agent1
    while terminal(board) == None:
        action = MMinimaxAgent(board, agent)

        board = result(board, agent, action)
        visualize(board)
        agent = otherPlayer(agent)
    return terminal(board)

#MMinimaxStim(exampleBoard3, turn(exampleBoard3))
visualize(exampleBoard4)
#stimulation(exampleBoard3, MMinimaxAgent(exampleBoard3, turn(exampleBoard3))

```

```
play(exampleBoard4, M0minimaxAgent, M0minimaxAgent, 1)
```

## 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?

You can make a limited depth search and have it run for only the first couple of moves and then switch to a heuristic search. This way your agent will be able to make a move quickly at the start and then switch to a more sophisticated search.

In [ ]: `# Your code/ answer goes here.`

## Playtime [5 points]

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

In [471]: `# Your code/ answer goes here.`  
`playtime = empty_board((4,4))`  
`play(playtime, miniMaxAgent, randomPlayer, 100)`

Out[471]: {'player1 wins': 51, 'player2 wins': 0, 'ties': 49}

This agent would never lose. It would either win or tie. This is because the miniMax agent would block the random agent from winning.

## Task 4: Heuristic Alpha-Beta Tree Search

### Heuristic evaluation function [15 points]

Define and implement a heuristic evaluation function.

```
In [479]: #a heuristic that gives every space a value for how good it is to play there.
def heuristic(board, player):
    center = len(board[0])//2
    value = 0
    for i in range(len(board)):
        for j in range(len(board[0])):
            if board[i][j] == player:
                value += 1
            elif board[i][j] == otherPlayer(player):
                value -= 1
            if j == center:
                value += 1
            elif j == center-1 or j == center+1:
                value += 0.5
    return value
```

## 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 [506]: # Your code/ answer goes here.
#heuri

#agent that uses the heuristic function and a depth limit of 3 using miniMax
def depthLimitedMiniMaxAgent(board, player, h = False):
    return depthLimitedMiniMaxSearch(board, player)["move"]

def depthLimitedMiniMaxSearch(board, player h = False):
    value, moveChoice = depthLimitedMiniMaxMax(board, player, 6, -np.inf, np.
    return {"move": moveChoice, "value": value}

def depthLimitedMiniMaxMax(board, player, depth, alpha, beta, h = False):
    validMoves = availableMoves(board)
    validMoves.sort(key=lambda x: abs(x-len(board[0])//2))

    if depth == 0 or terminal(board) != None:
        return heuristic(board, player), None

    value, moveChoice = -np.inf, None
    for action in validMoves:
        value2, action2 = depthLimitedMiniMaxMin(result(board, player, action
        if value2 > value:
            value, moveChoice = value2, action
            alpha = max(alpha, value)
        if value >= beta:
            return value, moveChoice
    return value, moveChoice

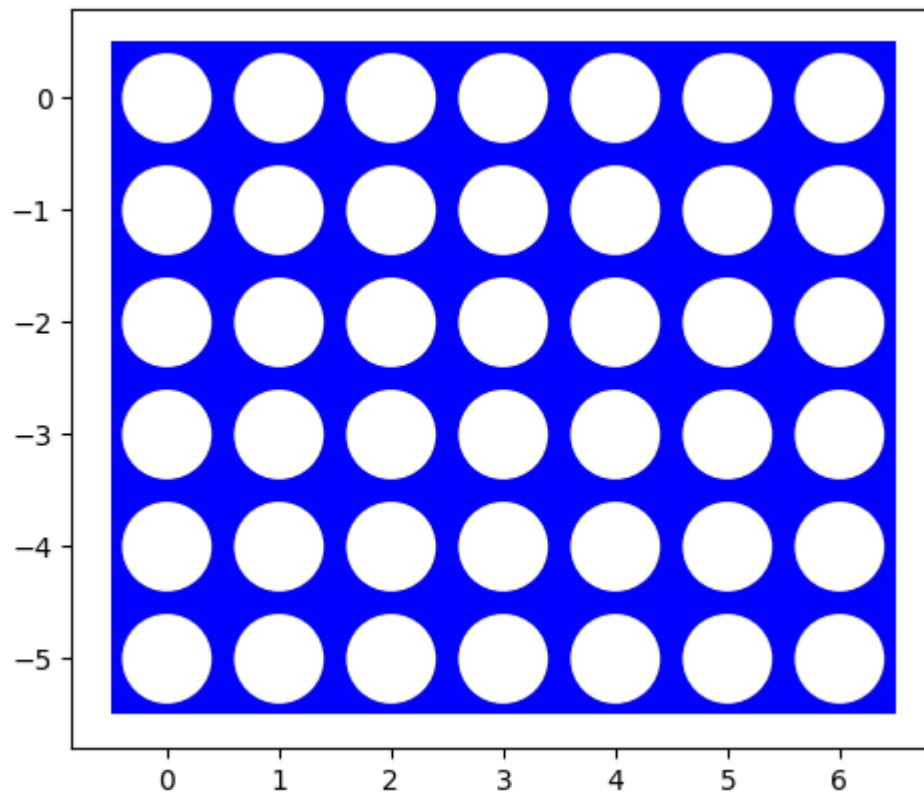
def depthLimitedMiniMaxMin(board, player, depth, alpha, beta, h = False):
    validMoves = availableMoves(board)
    validMoves.sort(key=lambda x: abs(x-len(board[0])//2))

    if depth == 0 or terminal(board) != None:
        return heuristic(board, player), None

    value, moveChoice = np.inf, None
    for action in validMoves:
        value2, action2 = depthLimitedMiniMaxMax(result(board, otherPlayer(pl
        if value2 < value:
            value, moveChoice = value2, action
            beta = min(beta, value)
        if value <= alpha:
            return value, moveChoice
    return value, moveChoice

board1 = empty_board()
visualize(board1)
play (board1, depthLimitedMiniMaxAgent, randomPlayer, 100)

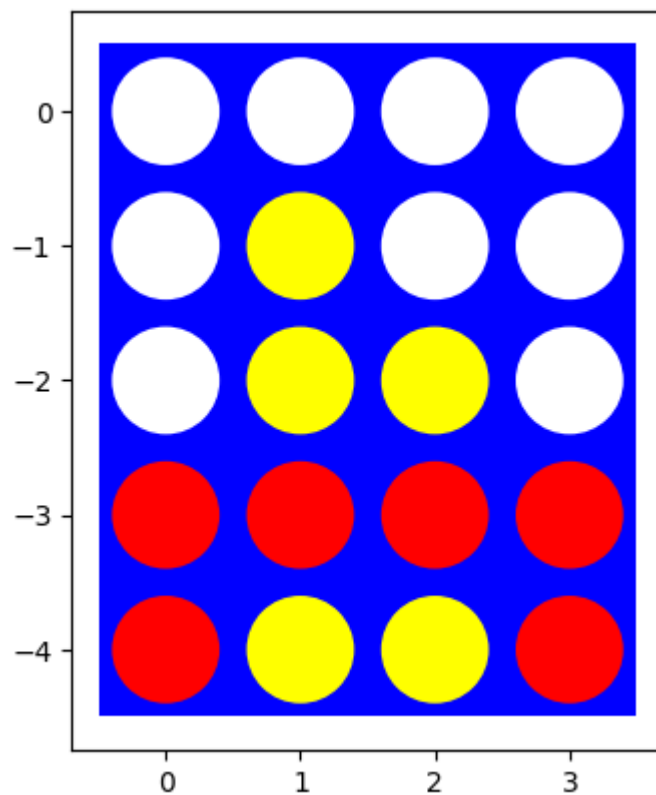
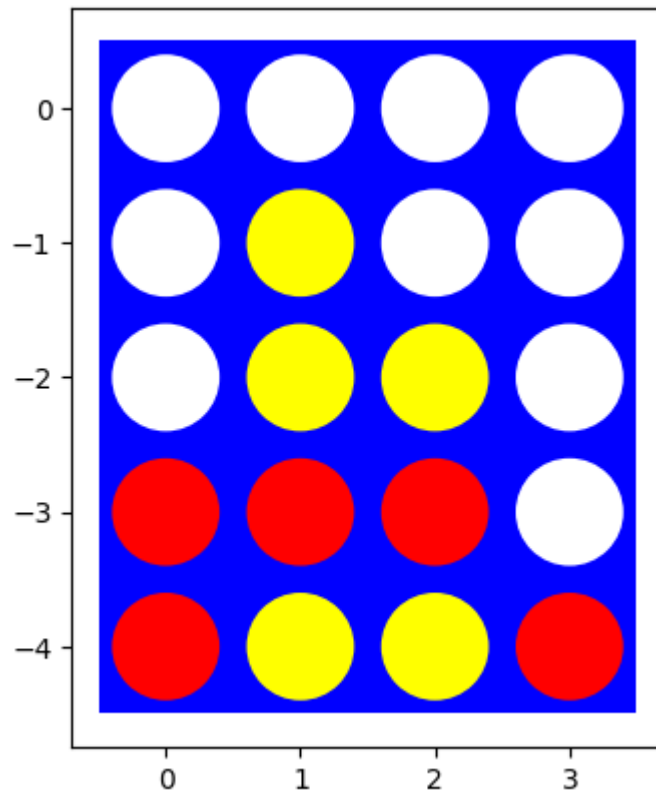
```



Out[506]: {'player1 wins': 52, 'player2 wins': 29, 'ties': 19}

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

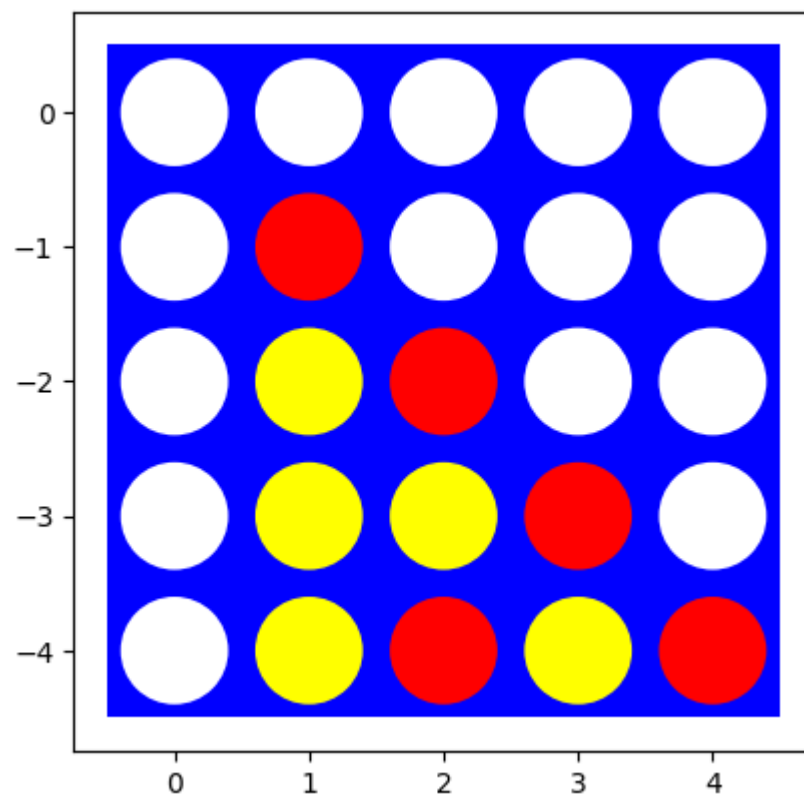
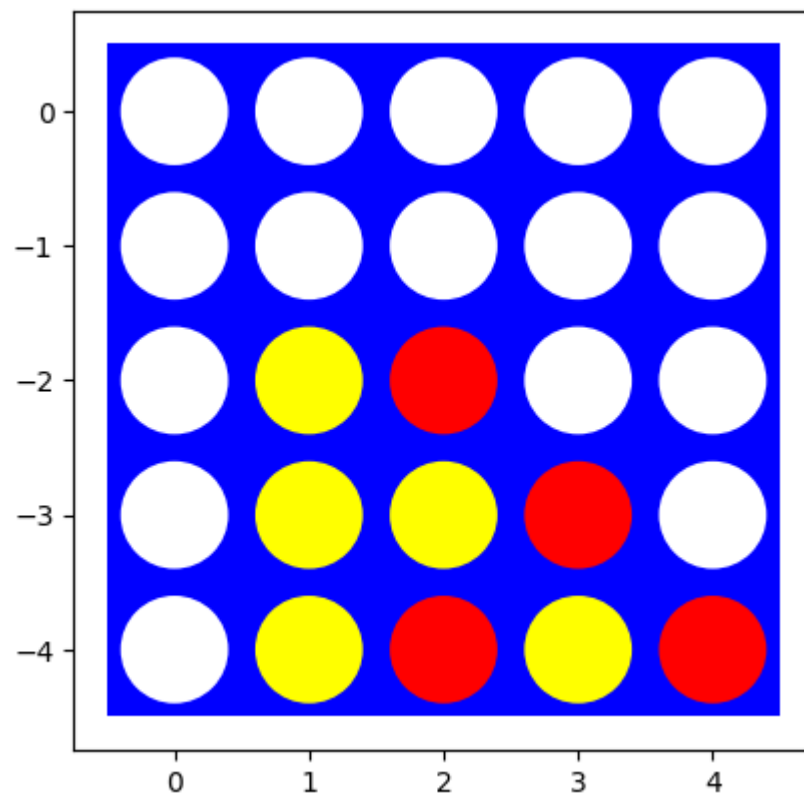
```
In [507]: # Your code/ answer goes here.  
visualize(exampleBoard1)  
play(exampleBoard1, depthLimitedMiniMaxAgent, randomPlayer, 1, True)
```



Out[507]: 1



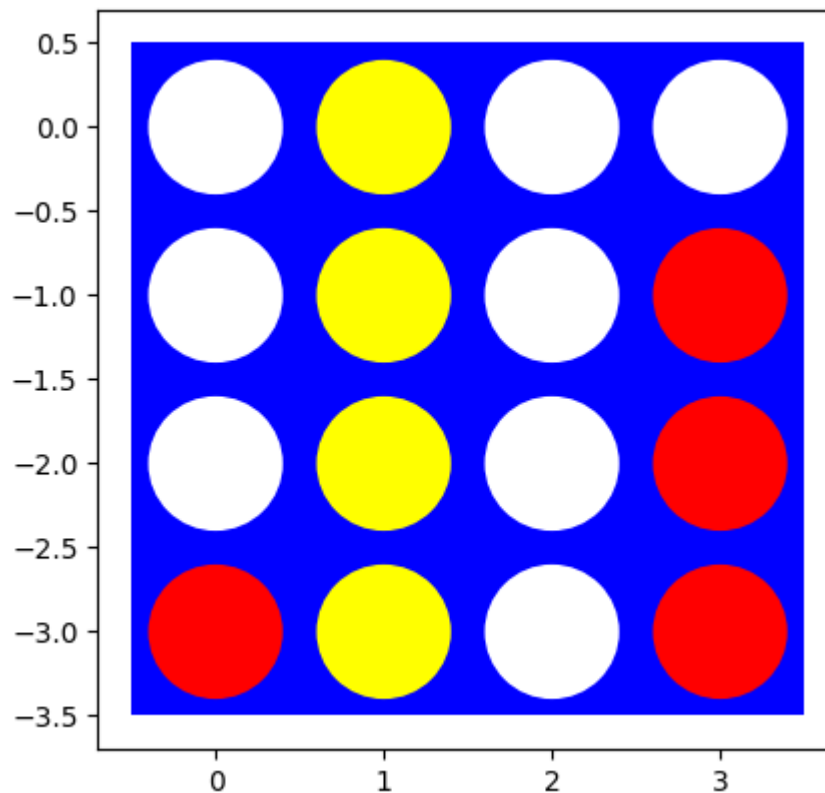
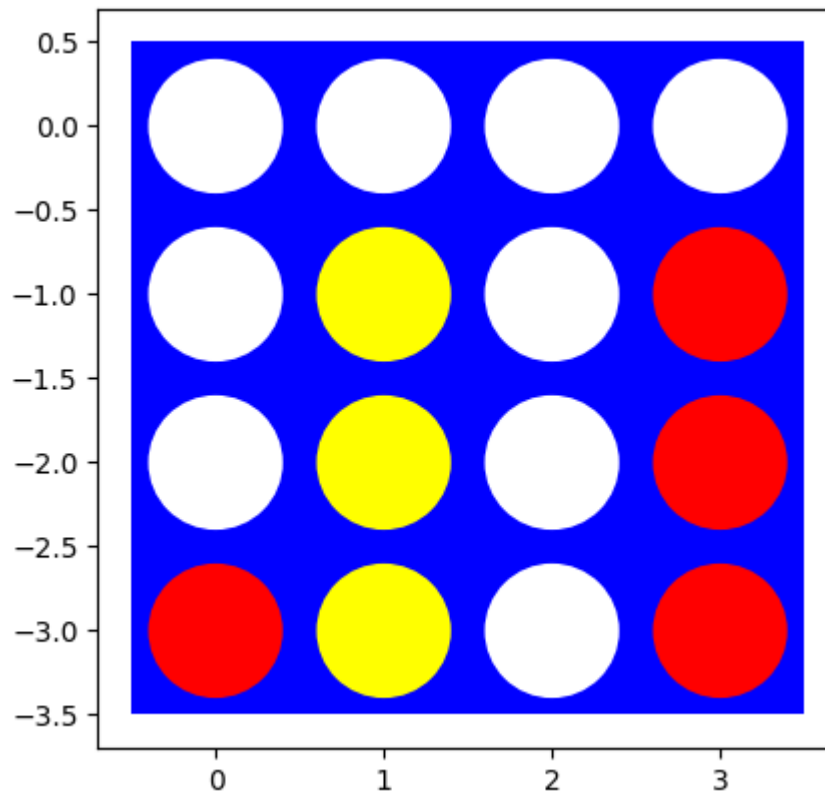
```
In [508]: visualize(exampleBoard2)  
play(exampleBoard2, depthLimitedMiniMaxAgent, randomPlayer, 1, True)
```



Out[508]: 1



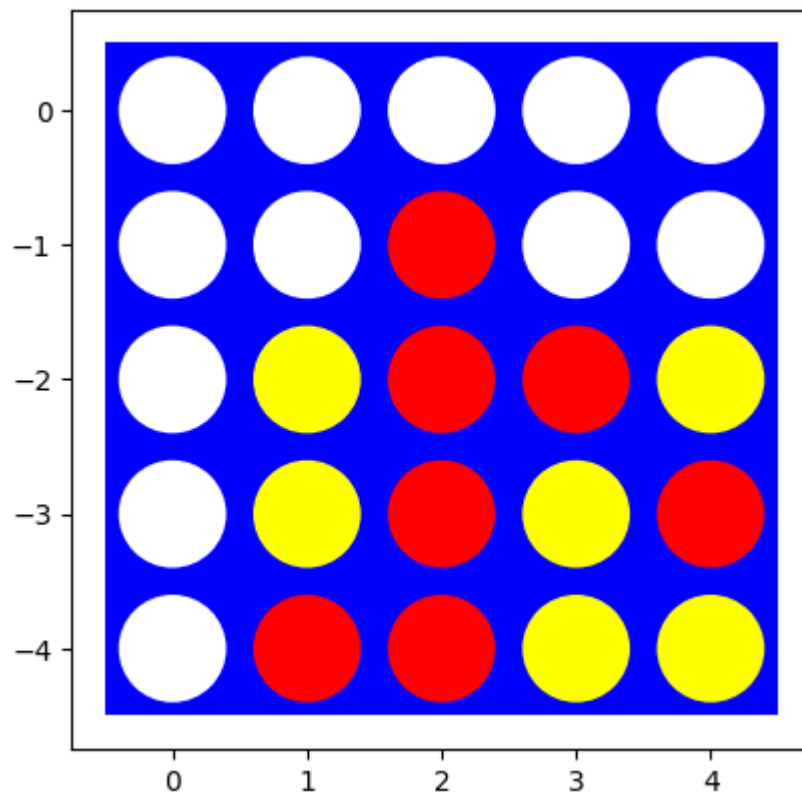
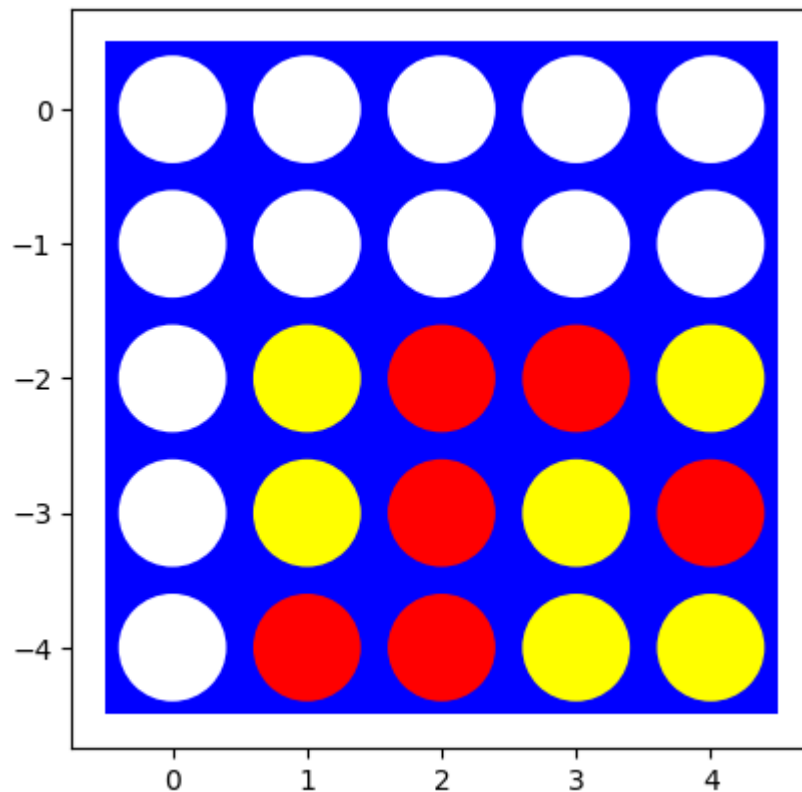
```
In [509]: visualize(exampleBoard3)  
play(exampleBoard3, depthLimitedMiniMaxAgent, randomPlayer, 1, True)
```



Out[509]: -1

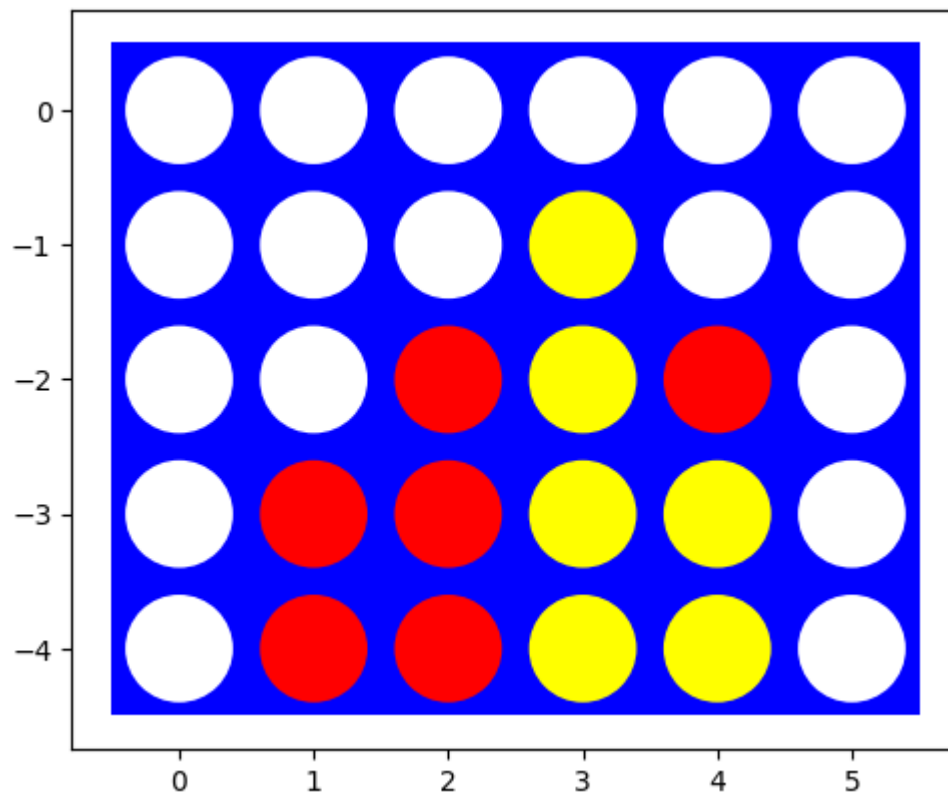
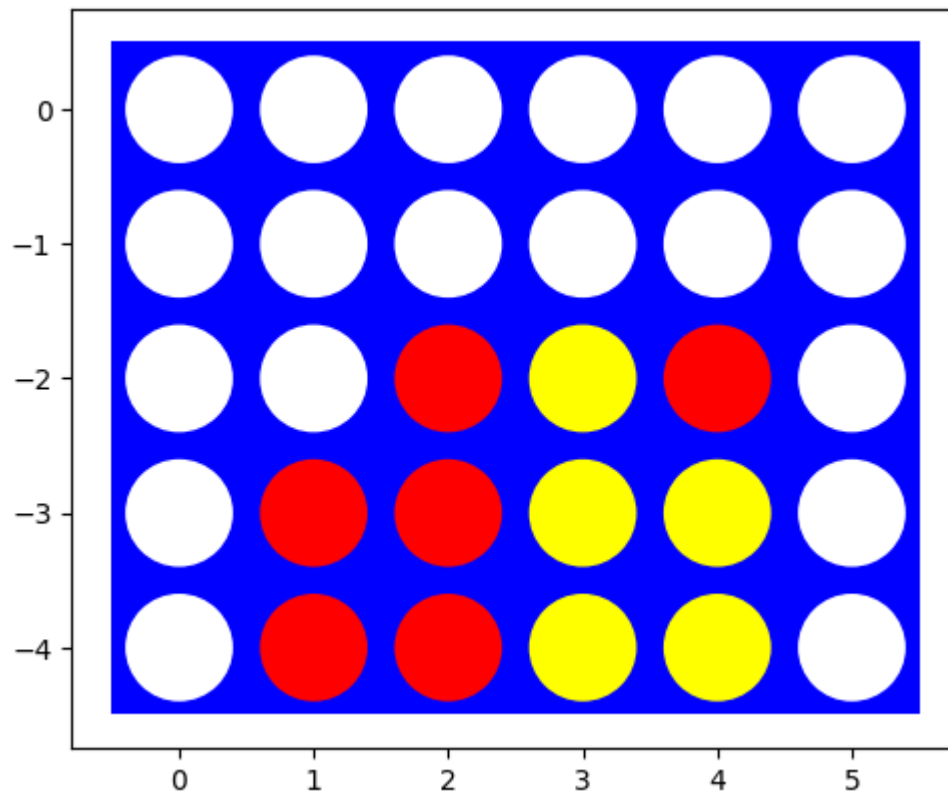


```
In [510]: visualize(exampleBoard4)  
play(exampleBoard4, depthLimitedMiniMaxAgent, randomPlayer, 1, True)
```



Out[510]: 1

```
In [511]: visualize(exampleBoard5)  
play(exampleBoard5, depthLimitedMiniMaxAgent, randomPlayer, 1, True)
```



Out[511]: -1

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.

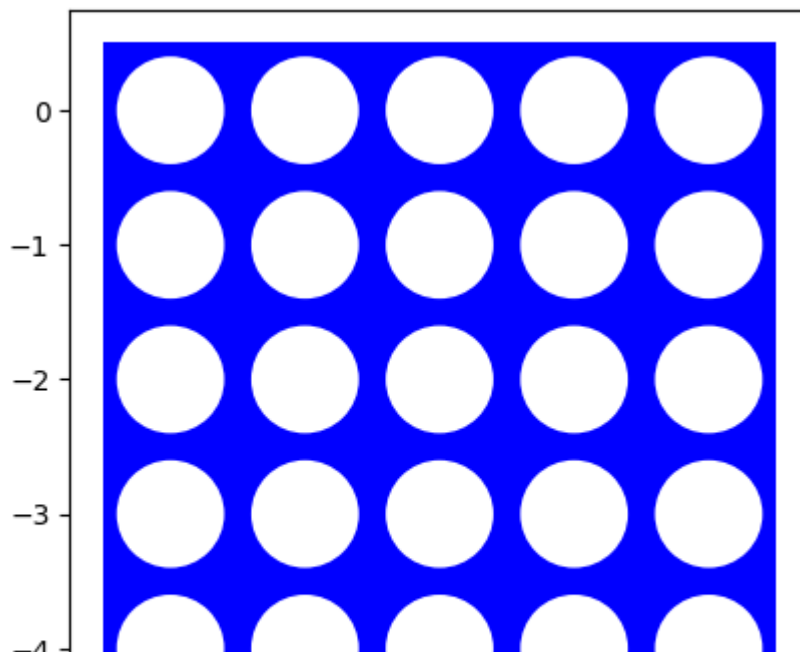
When adding the limit, the code runs significantly faster. in Test 5 especially, the code run 10 times faster at a limit of 6.

This code also fixed the bug I had with my base miniMax search. Yellow will now go for the obvious win which was not present in the first 5 tests.

## 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 [519]: ▶ # Your code/ answer goes here.  
board2 = empty_board((5,5))  
visualize(board2)  
  
play (board2, depthLimitedMiniMaxAgent, miniMaxAgent, 1, True)
```



After running my heuristic search agent against the original miniMax agent, It was unable to come to an answer in a reasonable amount of time. I believe this is because the original miniMax agent is able to search deeper than the heuristic agent. Therefore I was unable to get a result

## 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.

## 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.

In [ ]:  *# Your code/ answer goes here.*

### 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.

In [ ]:  *# Your code/ answer goes here.*