Adversarial Search: Playing Connect 4

Student Name: Blake William Gebhardt

I have used the following AI tools: Probably ChatGPT to help with syntax and logic errors.

I understand that my submission needs to be my own work: BwG

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)

Initial State: We start with an empty Connect-4 board. According to the Connect-4 wikipedia (and the blurb above), the standard board is 7x6, but "size variations include 5×4, 6×5, 8×7, 9×7, 10×7, 8×8, Infinite Connect-Four, and Cylinder-Infinite Connect-Four." We'll keep it simple with the 7x6.

Actions: Players can select a column to drop their piece into, and the piece will now occupy the first open position in the column.

Transition Model: The result function updates the game board by placing the player's disc in the column that they pick and makes sure the piece moves to the first open spot. It then should make sure the turn swaps to the other player.

Goal State: The goal state is obviously a state that has 4 of my pieces in a row while also not having 4 opponent pieces in a row. The 4 in a row can be horizontal, vertical, or diagonal. It's also possible that the board fills up with no 4s in a row, which would be a tie. We obsiously want to work to make moves that get us closer to having 4 in a row while minimizing the length of connected pieces the opponent has.

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

The state space of a 7x6 Connect 4 board is 4,531,985,219,092 total possible positions. I ripped this straight from the Wikipedia. There are 7x6 spaces (42 spaces). Each space can be empty, red, or yellow (3 choices), but there must be an equal amount of red/yellow or one more red than yellow. 3^42 is 1.0941899e+20, so it's definitely less than that. According to https://oeis.org/A212693, the sequence of valid boards after n-plays is as follows: 1, 7, 49, 238, 1120, 4263, 16422, 54859, 184275, 558186, 1662623, 4568683, 12236101, 30929111, 75437595, 176541259, 394591391, 858218743, 1763883894, 3568259802, 6746155945, 12673345045, 22010823988, 38263228189, 60830813459, 97266114959, 140728569039. The sum of these is 4,531,985,219,092.

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

Minimax search will have to go through a huge game tree. It will roughly be 7^42, since there are 7 possible moves at every turn the player has.

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

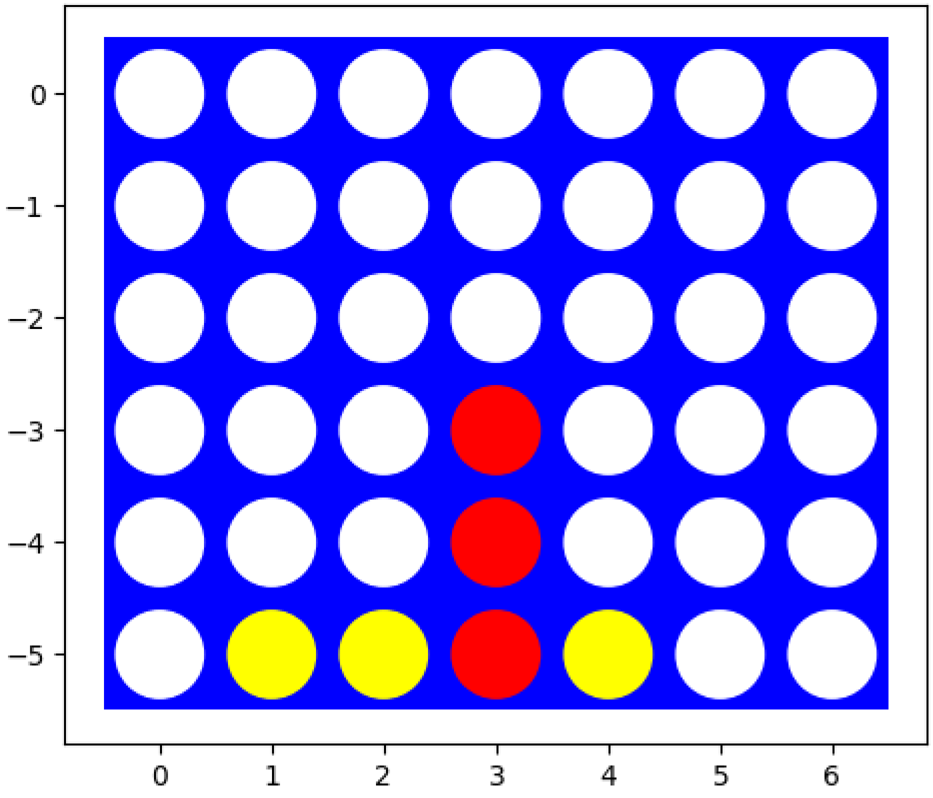
Use a numpy character array as the board.

```
In [ ]: import numpy as np

        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 [ ]:
```
# 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(bo
    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).

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

from timeit import default_timer as timer
import math
from scipy.signal import convolve2d #help me rotate a bunch of stuff fast

class ConnectHelp:
    @staticmethod
    def place(choice: int, board, player: int = None):
        curr_board = np.array(board).transpose()
        if player is None:
            arr_sum = curr_board.sum()
            if arr_sum > 0:
                player = -1
            else:
                player = 1

        lst = curr_board[choice]
        gen = (len(lst) - 1 - i for i, v in enumerate(reversed(lst)) if v
        last_idx = next(gen, None)
        curr_board[choice][last_idx] = player
        return curr_board.transpose()
    @staticmethod
    def get_valid_moves(board):
        return [i for i, j in enumerate(board[0]) if j == 0]

    @staticmethod
    def to_move(board):
        curr_board = np.array(board)
        arr_sum = curr_board.sum()
        if arr_sum > 0:
            player = -1
        else:
            player = 1
        return player

    @staticmethod
    def check_win(board):
        # Check if a draw
        if len(ConnectHelp.get_valid_moves(board)) == 0:
            return None

        # winning things
        horizontal_win = np.array([[1, 1, 1, 1]])
        vertical_win = np.transpose(horizontal_win)  # Transpose of the h
        positive_diag = np.eye(4, dtype=np.uint8)  # (top-left to bottom-
        negative_diag = np.fliplr(positive_diag)  # (top-right to bottom-


        win_list = [horizontal_win, vertical_win, positive_diag, negative

        # Iterate through each detection kernel to check for winning patt
        for win in win_list:
            # Use convolution to detect patterns on the board using the c
            a = convolve2d(board, win, mode='valid')

            # Check if a winning pattern for Player 1 (sum equals 4) is f
            if (a == 4).any():
                return 1
            # Check if a winning pattern for Player 2 (sum equals -4) is 
            if (a == -4).any():
                return -1

        # no win, continue game
        return 0


    @staticmethod
    def evaluate_board(board, player):
        try:
            return ConnectHelp.calc_utility(player, board) * 2, True
        except:
            # winning things
            horizontal_win = np.array([[1, 1, 1, 1]])
            vertical_win = np.transpose(horizontal_win)  # Transpose of t
            positive_diag = np.eye(4, dtype=np.uint8)  # (top-left to bot
            negative_diag = np.fliplr(positive_diag)  # (top-right to bot


            win_list = [horizontal_win, vertical_win, positive_diag, nega
            utils = []
            for win in win_list:
                a = convolve2d(board, win, mode='valid')
                if (a == (3 * player)).any():
                    times = np.count_nonzero(a == (3 * player))
                    utils += [0.5] * times
                if (a == (2 * player)).any():
                    times = np.count_nonzero(a == (2 * player))
                    utils += [0.25] * times
                if (a == (3 * player) * -1).any():
                    times = np.count_nonzero(a == (3 * player * -1))
                    utils += [-0.5] * times
                if (a == (2 * player) * -1).any():
                    times = np.count_nonzero(a == (2 * player * -1))
                    utils += [-0.25] * times

            #a winning strategy in connect4 is to make a 7 shape
            #let's give massive utility for making a 7 and take away a lo
            forward_seven = np.array([
                [1, 1, 1],
                [0, 1, 0],
                [1, 0, 0]
            ])
            seven_traps = [forward_seven]
            seven_traps.append(np.array([
                [1, 1, 1],
                [0, 1, 0],
                [0, 0, 1]
            ]))
            seven_traps = [np.flip(arr) for arr in seven_traps]
            for trap in seven_traps:
                a = convolve2d(board, trap, mode='valid')
                if (a == (5 * player)).any():
                    times = np.count_nonzero(a == (5 * player))
                    utils += [0.75] * times
                if (a == (5 * player * -1)).any():
                    times = np.count_nonzero(a == (5 * player * -1))
                    utils += [-0.75] * times

            if len(utils) != 0:
                return ConnectHelp.sigmoid(np.sum(utils)), False
            else:
                return 0, False

    @staticmethod
    def time_function(theFunc, *args):
        start = timer()
        theFunc(*args)
        print(f"{theFunc.__name__}: {(timer() - start) * 1000}")

    @staticmethod
    def calc_utility(player, board):
        winner = ConnectHelp.check_win(board)
        if len(ConnectHelp.get_valid_moves(board)) == 0:
            return 0
        if winner == 0:
            raise Exception("state won't terminate")
        if winner == player:
            return 1
        else:
            return -1
    @staticmethod
    def sigmoid(x):
        return ((1 / (1 + math.e ** -x)) * 2) - 1
```

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 [ ]: # Your code/ answer goes here.
        import random
        def random_player(board,player= None):
            try:
                return {"move":random.choice(ConnectHelp.get_valid_moves(board)),
            except:
                return None
```

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 [ ]: # Your code/ answer goes here.
        def random_environment(num_iters=1000):
            players = [1,-1]
            attempts = []
            for _ in range(num_iters):
                board = empty_board()
                turn_num = 0
                while(ConnectHelp.check_win(board) == 0):

                    the_choice = random_player(board)
                    if(the_choice is None): break
                    else: the_choice = the_choice['move']
                    turn = players[turn_num % len(players)]
                    board = ConnectHelp.place(the_choice,board,player=turn)
                    turn_num += 1

                attempts.append({
                    "turns_taken":turn_num,
                    "winner":ConnectHelp.check_win(board)
                })
            return attempts


        def print_results(attempts):
            p1_wins = 0
            p2_wins = 0
            ties = 0
            total_turns = 0
```

```
    for val in attempts:
        if val["winner"] == 1:
            p1_wins += 1
        elif val["winner"] == -1:
            p2_wins += 1
        elif val["winner"] is None:
            ties += 1

        total_turns += val["turns_taken"]

    num_attempts = len(attempts)
    if num_attempts > 0:
        average_turns = total_turns / num_attempts
    else:
        average_turns = 0

    print(f'''
    Player 1 Wins: {p1_wins}
    Player 2 Wins: {p2_wins}
    Ties: {ties}
    Average Turns: {average_turns}''')




attempts = random_environment(num_iters=1000)
print_results(attempts)
```

```
    Player 1 Wins: 577
    Player 2 Wins: 419
    Ties: 4
    Average Turns: 21.287
```

It seems that Player 1 has a slight advantage when playing randomly. This is about expected, as generally going first in any game puts you at a significant advantage. Other than that, it's about evenly matched.

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.

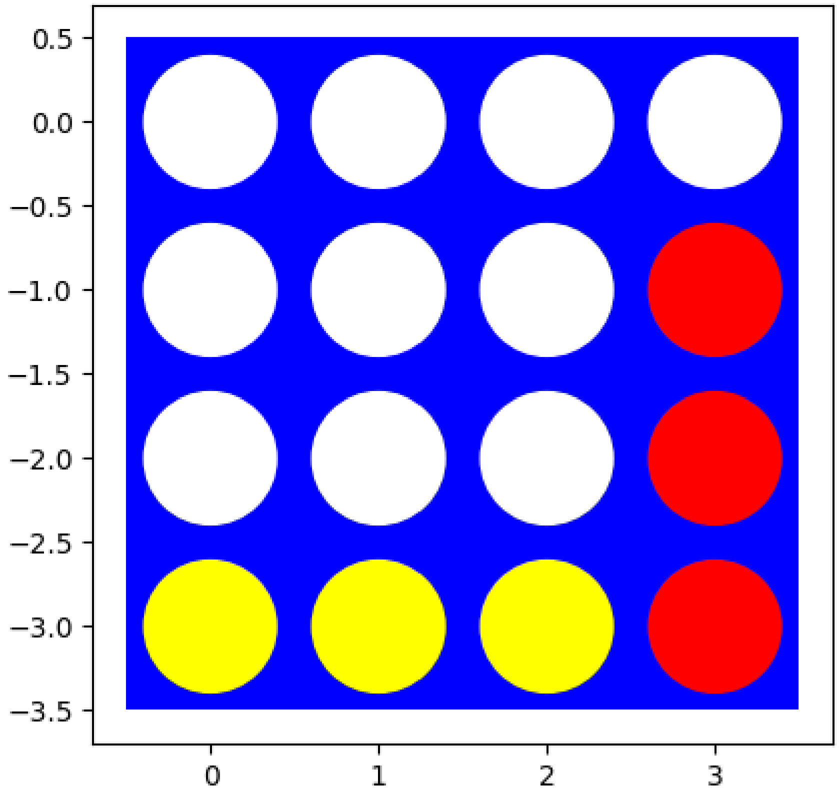
- The search space for a 6×7 board is large. You can experiment with smaller boards (the smallest is 4×4) and/or changing the winning rule to connect 3 instead of 4.

In [ ]:
```python
ITERATIONS = 0
DEBUG = False

def min_value(board, player, alpha, beta):
    global ITERATIONS
    ITERATIONS += 1

    try:
        return ConnectHelp.calc_utility(player, board), None
    except:
        v = float('inf')
        for action in ConnectHelp.get_valid_moves(board):
            v2, a2 = max_value(ConnectHelp.place(action, board), player,
            if v2 < v:
                v = v2
                move = action
                beta = min([beta, v])
            if v <= alpha:
                if DEBUG >= 2: print("going back")
                return v, move
        return v, move

def max_value(board, player, alpha, beta):
    global ITERATIONS
    ITERATIONS += 1
    try:
        return ConnectHelp.calc_utility(player, board), None
    except:
        v = float('-inf')
        for action in ConnectHelp.get_valid_moves(board):
            v2, a2 = min_value(ConnectHelp.place(action, board), player,
            if v2 > v:
                v = v2
                move = action
                alpha = max([alpha, v])
            if v >= beta:
                if DEBUG >= 2: print("going back")
                return v, move
        return v, move

def minimax_search(board, player=None):
    global ITERATIONS
    ITERATIONS = 0
    if player is None:
        player = ConnectHelp.to_move(board)
    board = np.array(board)
    if np.count_nonzero(board == player, axis=None) < 2:
        return {"move": random.choice(ConnectHelp.get_valid_moves(board))
    value, move = max_value(board, player, alpha=float('-inf'), beta=floa
    color = "Red" if player == 1 else "Yellow"
    global DEBUG
    if DEBUG:
        print(f"Player: {color} | value: {value} | move: {move}")
    return {"move": move, "value": value}

def minimax_env(board=None, verbose=False):
    if board is None:
        board = empty_board(shape=(4, 4))
    global DEBUG
    DEBUG = verbose
    while ConnectHelp.check_win(board) == 0:
        board = ConnectHelp.place(minimax_search(board)["move"], board)
    visualize(board)
```

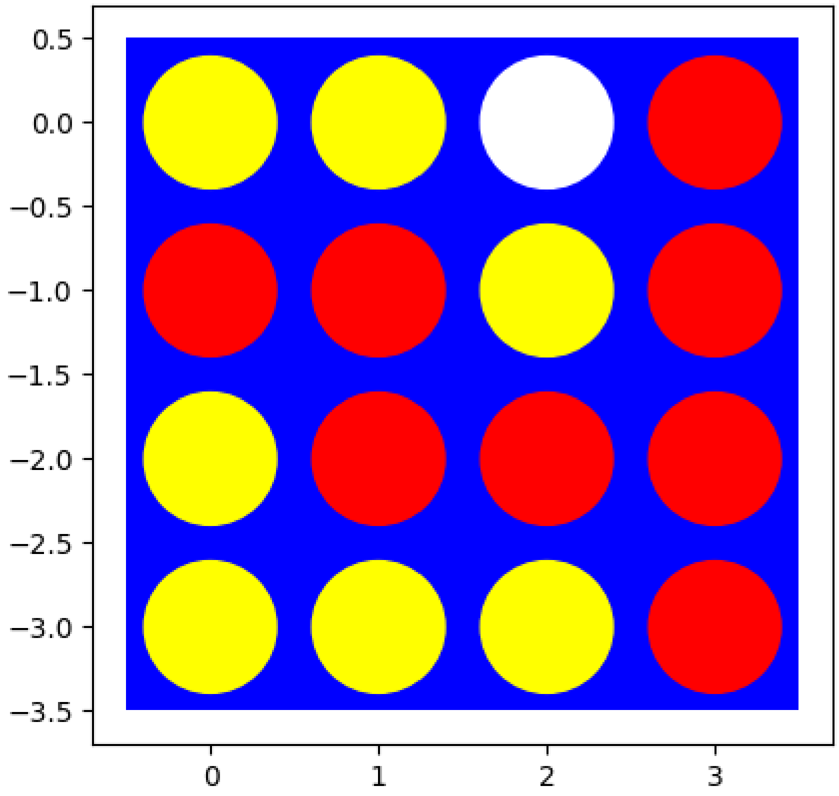
Experiment with some manually created boards (at least 5) to check if the agent spots winning opportunities.

```
In [ ]: board1 = [[0,0,0,0],
                  [0,0,0,1],
                  [0,0,0,1],
                  [-1,-1,-1,1]]
        board2 = [[0,0,0,-1],
                  [-1,1,0,1],
                  [1,-1,1,1],
                  [-1,-1,-1,1]]
        board3 = [[0,0,0,1],
                  [0,0,0,1],
                  [0,0,1,1],
                  [0,-1,-1,-1]]
        board4 = [[0,0,0,-1],
                  [0,1,0,1],
                  [0,1,-1,1],
                  [0,-1,-1,1]]
        board5 = [[0,0,0,0],
                  [0,0,0,1],
                  [0,0,0,1],
                  [0,-1,-1,1]]
        boards = [board1,board2,board3,board4,board5]


        for board in boards:
            print("BEFORE MINIMAX")
            visualize(board)
            print("AFTER MINIMAX")
            minimax_env(board)
```

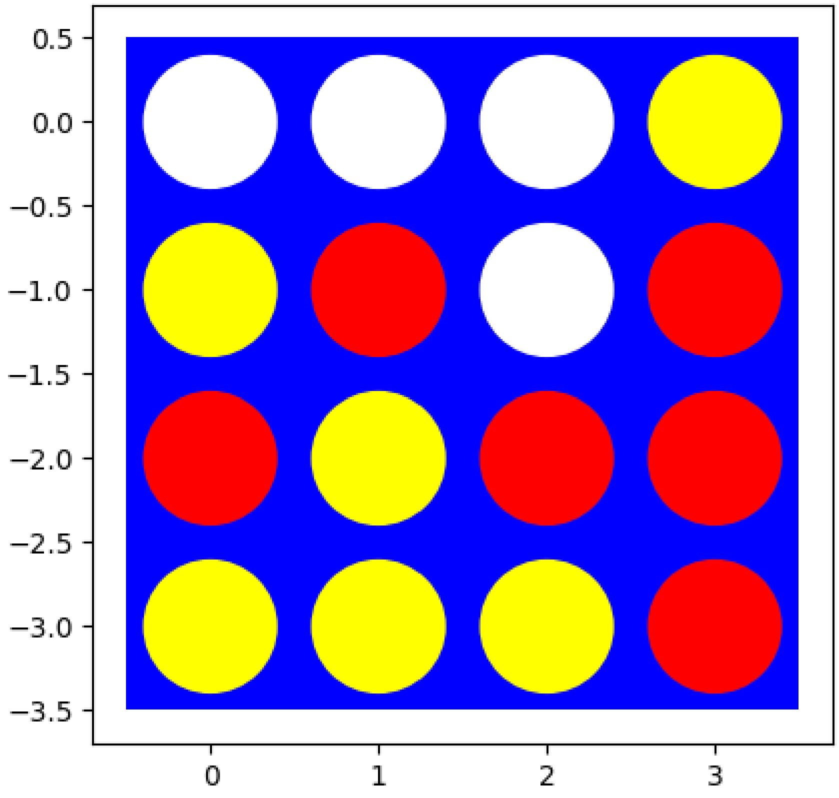
```
BEFORE MINIMAX
```



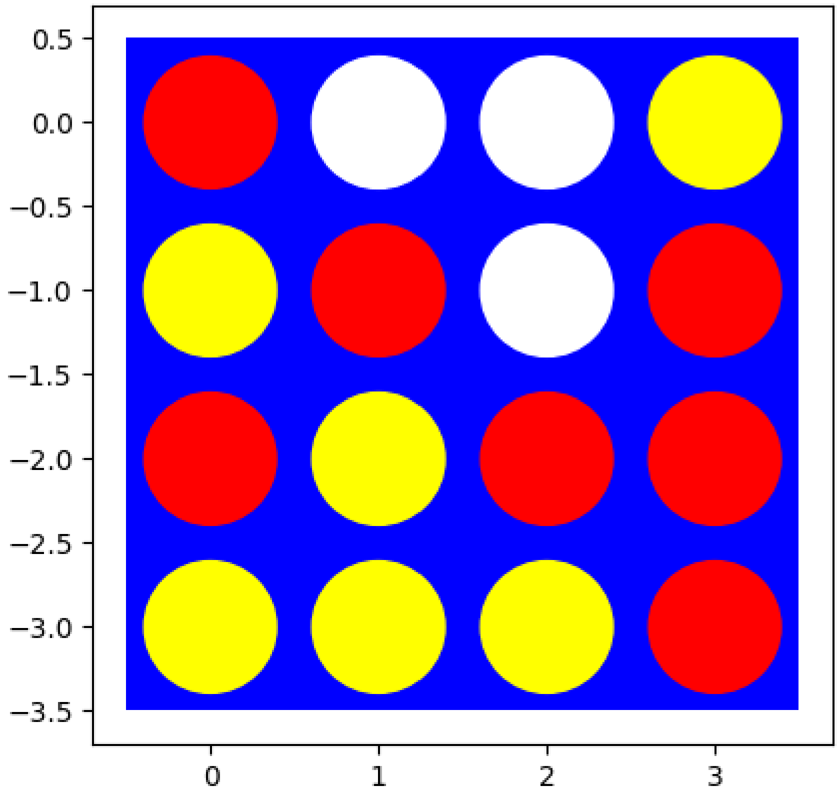
```
AFTER MINIMAX
```



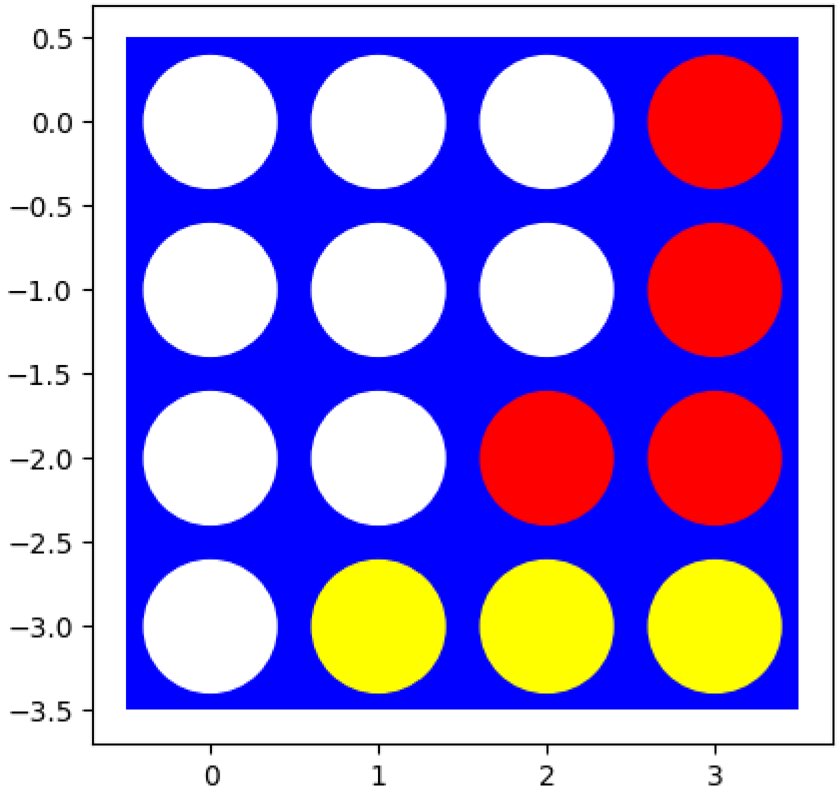
```
BEFORE MINIMAX
```



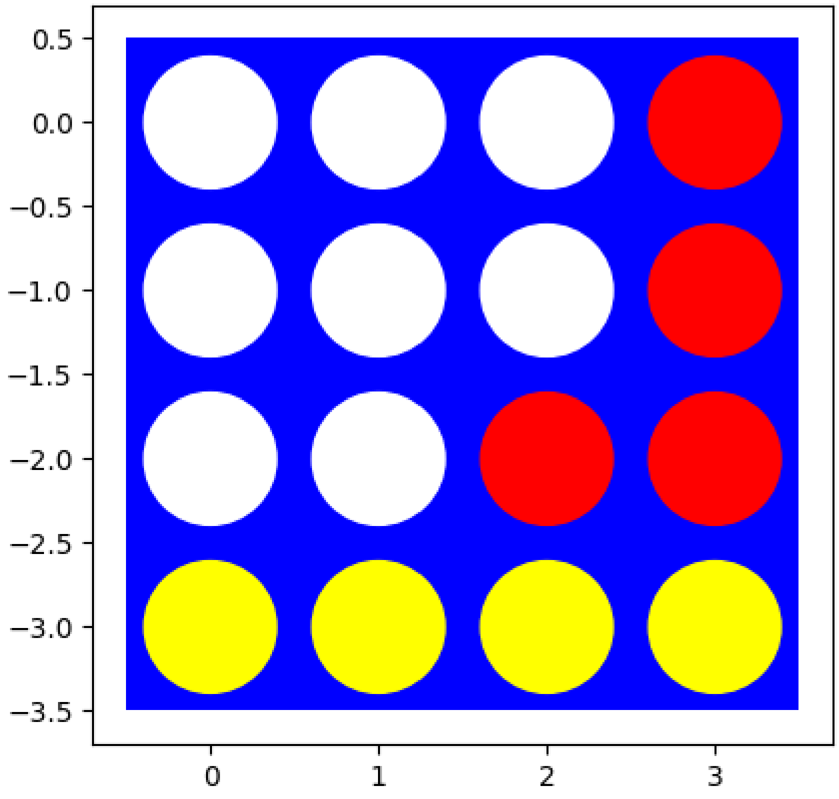
```
AFTER MINIMAX
```



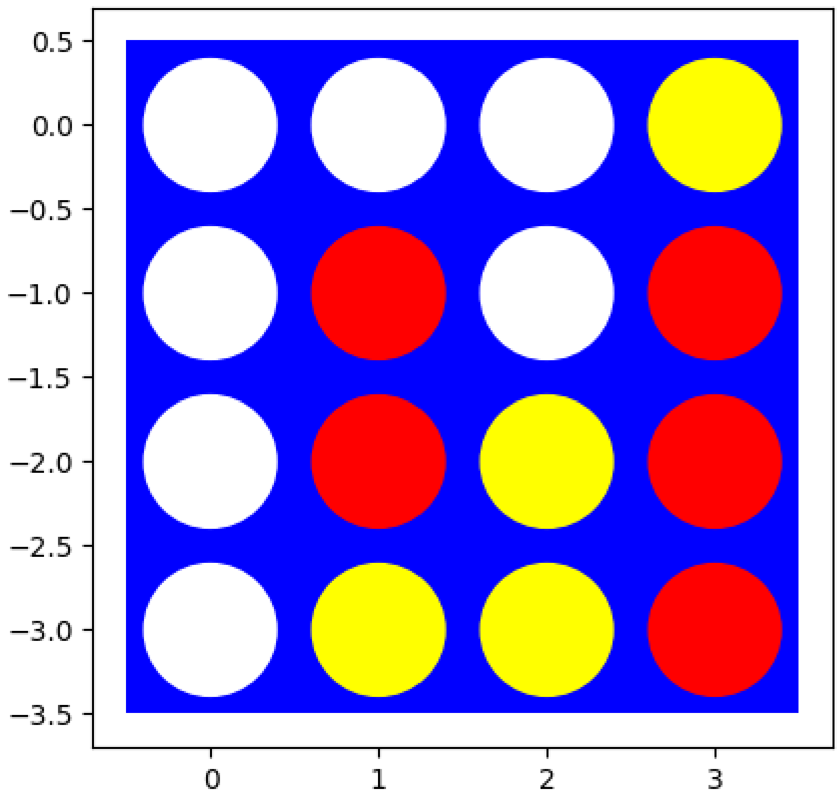
```
BEFORE MINIMAX
```



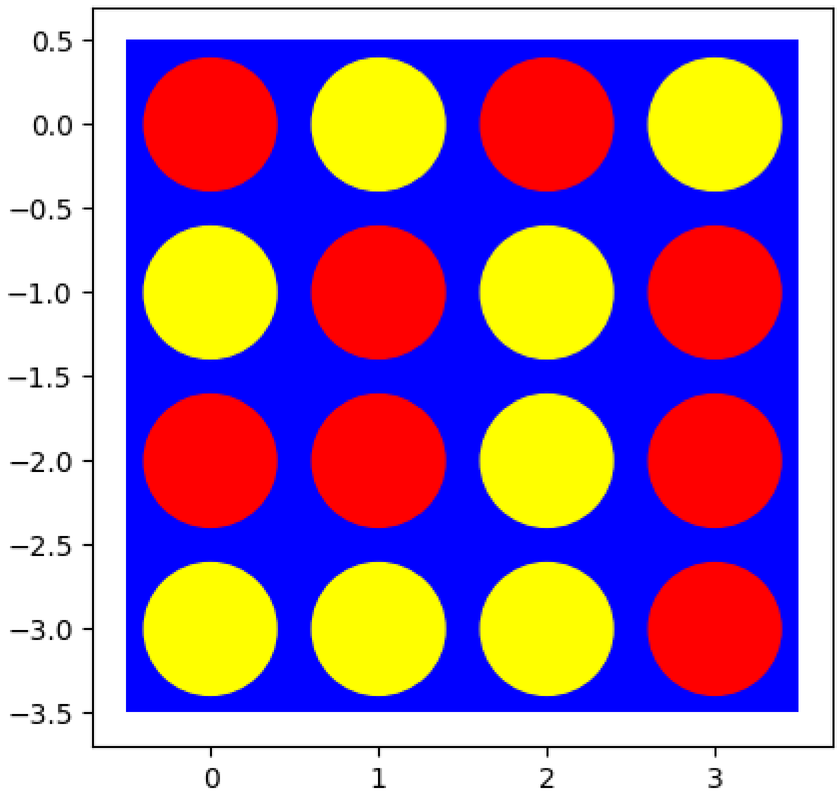
```
AFTER MINIMAX
```



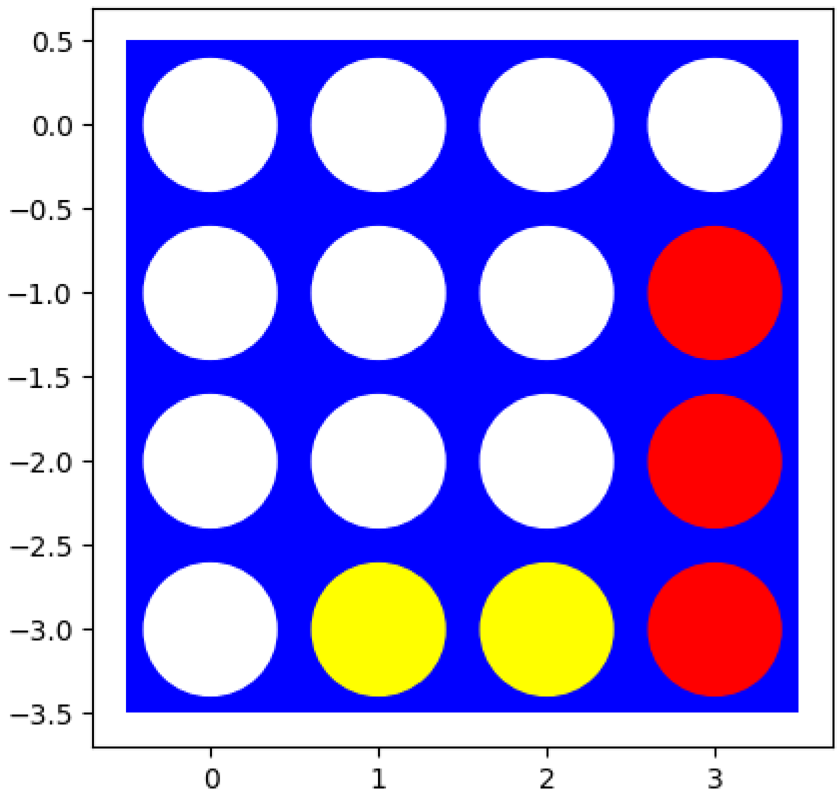
```
BEFORE MINIMAX
```



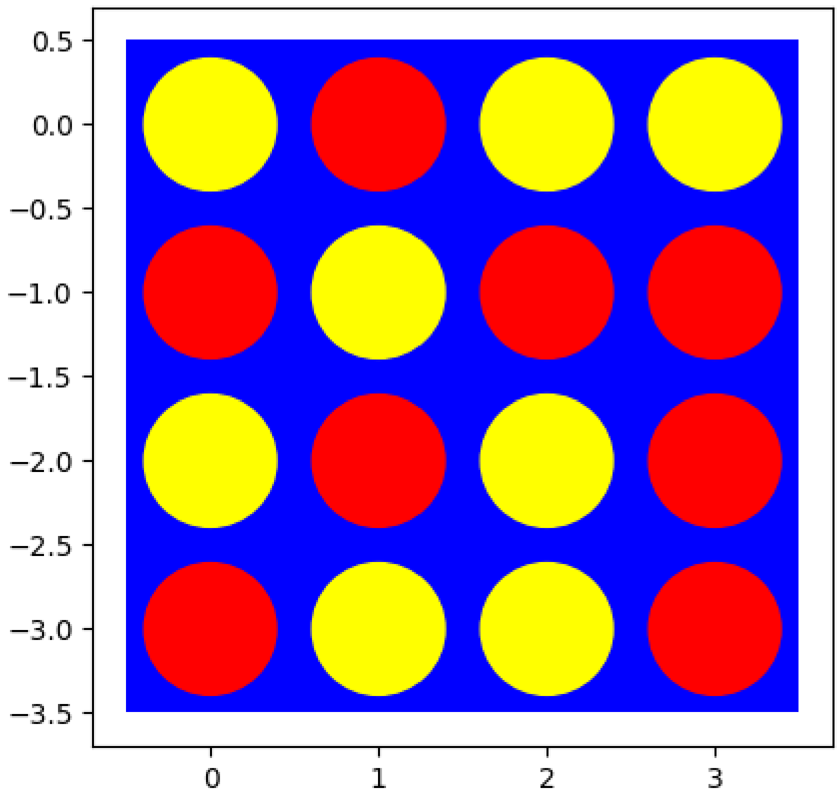
```
AFTER MINIMAX
```



```
BEFORE MINIMAX
```



```
AFTER MINIMAX
```

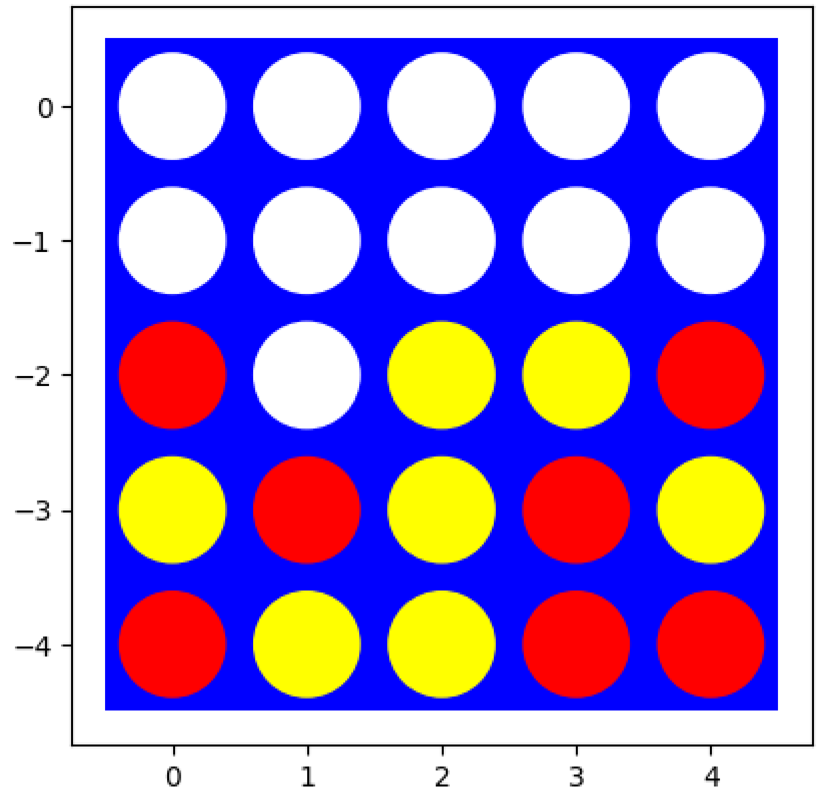


How long does it take to make a move? Start with a smaller board with 4 columns and make the board larger by adding columns.

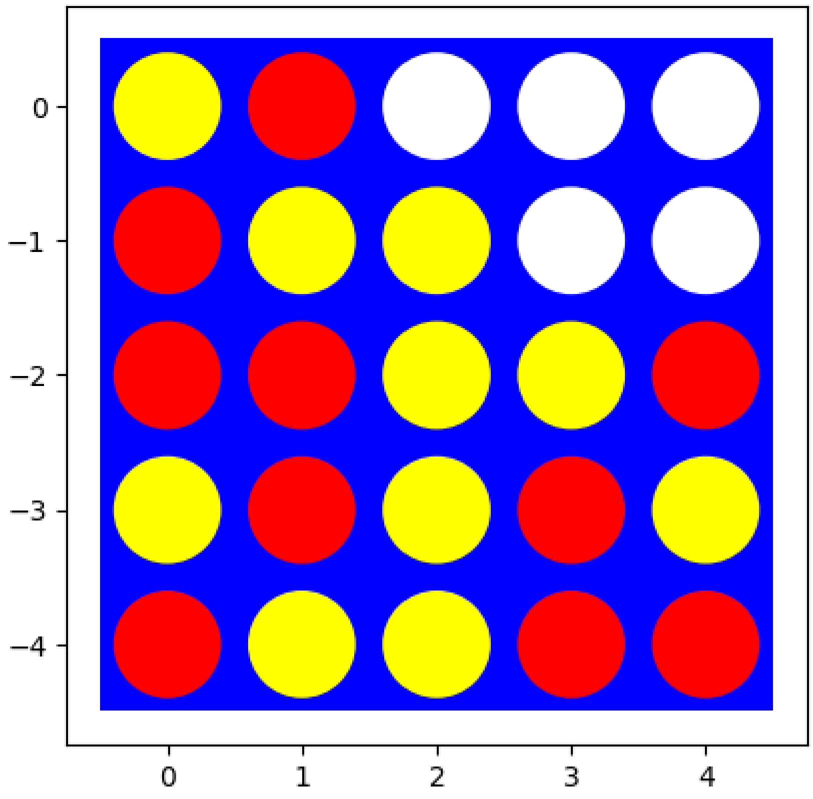
It's a fairly short time to solve each of the 5 boards, roughly 1 second each for the 4 column boards. Let me try a larger board.

```
In [ ]: # Your code/ answer goes here.
        board = [ [0,0,0,0,0],
                  [0,0,0,0,0],
                  [1,0,-1,-1,1],
                  [-1,1,-1,1,-1],
                  [1,-1,-1,1,1]]

        print("BEFORE")
        visualize(board)
        print("AFTER")
        minimax_env(board)
```

```
BEFORE
```



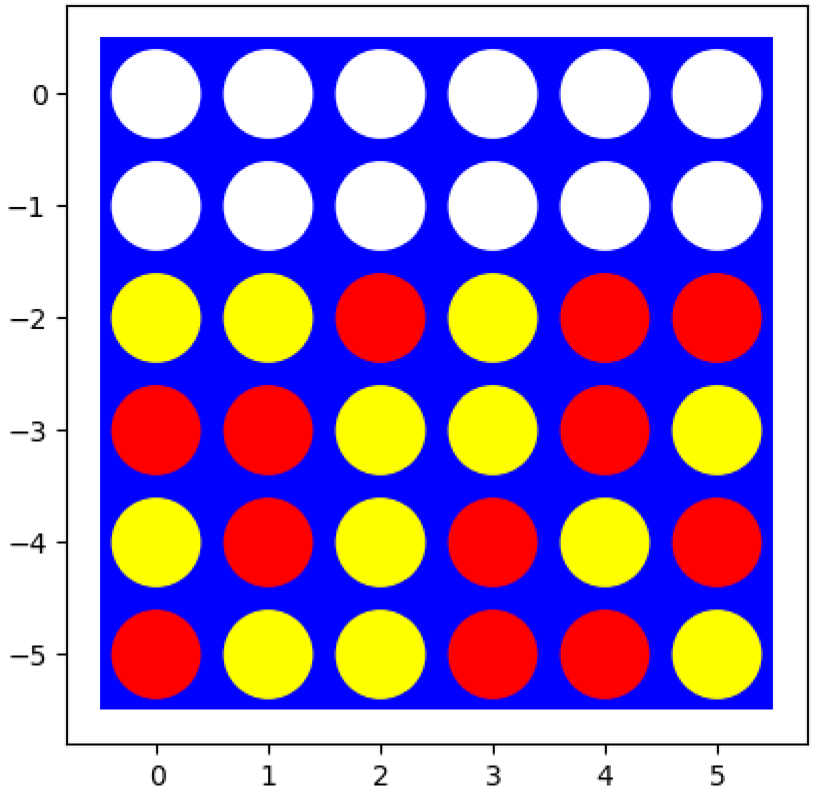
```
AFTER
```



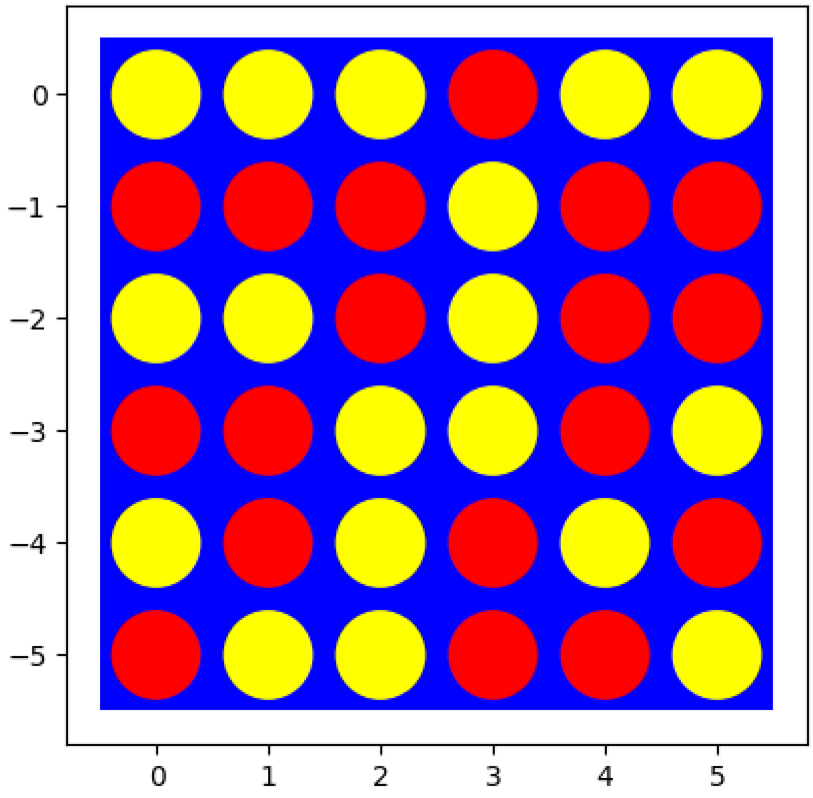
One 5x5 took 5 seconds by itself. I'll try a 6x6, I'm going to assume that it will take took long to be efficient.

```
In [ ]: board = [ [0,0,0,0,0,0],
                  [0,0,0,0,0,0],
                  [-1,-1,1,-1,1,1],
                  [1,1,-1,-1,1,-1],
                  [-1,1,-1,1,-1,1],
                  [1,-1,-1,1,1,-1]]

        print("BEFORE")
        visualize(board)
        print("AFTER")
        minimax_env(board)
```

```
BEFORE
```



AFTER



The 6x6 took 23 seconds or so. Much too slow.

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 [ ]:
```
# Your code/ answer goes here.
def initial_moves(board,player):
    if(np.count_nonzero(board == player,axis=None) == 0 and len(board[0])
        return {"move":0,"value":0}
    elif(np.count_nonzero(board == player,axis=None) == 0 and len(board[0
        return {"move":int(round(len(board[0]) / 2,0)),'value':0}
    elif(np.count_nonzero(board == player,axis=None) == 1 and len(board[0
        return {"move":len(board[0]) -1,'value':0}
    elif(np.count_nonzero(board == player,axis=None) == 1 and len(board[0
        return {'move':random.choice([int(round(len(board[0] / 2),0) - 1)
    else:
        return random.choice(ConnectHelp.get_valid_moves(board))

def minimax_search_optimal_first_move(board,player=None):
    global ITERATIONS
    ITERATIONS = 0
    if player is None:
        player = ConnectHelp.to_move(board)
    board = np.array(board)
    if(np.count_nonzero(board == player,axis=None) < 2):
        return initial_moves(board,player)


    value,move = max_value(board,player,alpha=float('-inf'),beta=float('i
    color = "Red"
    if player == -1: color = "Yellow"
    print(f"Player: {color} | value: {value} | move:{move}")
    return {"move":move,"value":value}
```

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?

I think just logically I would prefer to start in the dead middle of the board. In a normal board, the middle is generally the optimal choice. In a board that doesn't have a dead middle, my code should just randomly pick either the left middle or the right middle. After that, I will likely just pick another random move that is in the middle. Moves on the edges are generally not optimal, so we'll avoid those in the early stages.

Playtime [5 points]

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

In [ ]:
```
# Your code/ answer goes here.
#also adapted from the code you gave us

def dynamic_env(players,theBoard=None,verbose=False):
    if theBoard is None:
        theBoard= empty_board(shape=(4,5))
    turns = 0

    while(ConnectHelp.check_win(theBoard) == 0):
        player = players[turns % 2]['player']
        player_move = players[turns % 2]['algo'](theBoard,player=player)
        theBoard = ConnectHelp.place(player_move['move'],theBoard,player=

        if(verbose):
            print(f"Turn #{turns} | algo:{players[turns%2]['algo'].__name
            visualize(theBoard)
        turns +=1
    return {
        "winner":ConnectHelp.check_win(theBoard),
        "turns_taken":turns
    }

def dynamic_stats_env(players, num_iters=5):
    results = []
    times = []
    for i in range(num_iters):
        print(f"Iteration #{i}")
        start_time = timer()
        iteration_result = dynamic_env(players)
        end_time = timer()
        results.append(iteration_result)
        times.append(end_time - start_time)

    average_time = np.mean(times)
    print(f"Average time : {average_time}")
    print_results(results)
```

This code takes a while, but nearly every time the minimax search wins. The Players code is adapted from what you sent us so I can have agents play against each other.

In [ ]:
```
players = [
    {
        "algo":minimax_search_optimal_first_move,
        "player":1
     },
    {
        "algo":random_player,
        "player":-1
    }

]

#dynamic_env(players,verbose=False)
```

Task 4: Heuristic Alpha-Beta Tree Search

Heuristic evaluation function [15 points]

Define and implement a heuristic evaluation function.

The heuristic evaluation function (from up top) is repeated here:

```
...
            for kernel in detection_kernels:
                a = convolve2d(board, kernel,
mode='valid')
                if (a == (3 * player)).any():
                    times = np.count_nonzero(a == (3 *
player))
                    utils += [0.5] * times
                if (a == (2 * player)).any():
                    times = np.count_nonzero(a == (2 *
player))
                    utils += [0.25] * times
                if (a == (3 * player) * -1).any():
                    times = np.count_nonzero(a == (3 *
player * -1))
                    utils += [-0.5] * times
                if (a == (2 * player) * -1).any():
                    times = np.count_nonzero(a == (2 *
player * -1))
                    utils += [-0.25] * times

            if len(utils) != 0:
                return
HelperFunctions.sigmoid(np.sum(utils)), False
            else:
                return 0, False
    ...
```

This eval function uses numpy's convolve to look for rows of two or three and assigns proper utility ratings to each (either .5 or .25). If the opponent has these strings, the utility is negated.

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 [ ]:
```
# Your code/ answer goes here.
class Move:
    def __init__(self,action,util):
        self.action = action
        self.util = util
    def __lt__(self,other):
        return self.util < other.util
    def __gt__(self,other):
        return self.util > other.util
    def __le__(self,other):
        return self.util <= other.util
    def __ge__(self,other):
        return self.util >= other.util

def order_actions_on_util(actions,util_func,player,board):
    set_of_moves = [(move,util_func(ConnectHelp.place(move,board,player),
    set_of_moves.sort(key = lambda x: x[1],reverse=True)
    return [move[0] for move in set_of_moves]


diditwork = order_actions_on_util(ConnectHelp.get_valid_moves(empty_board
print(diditwork)
```

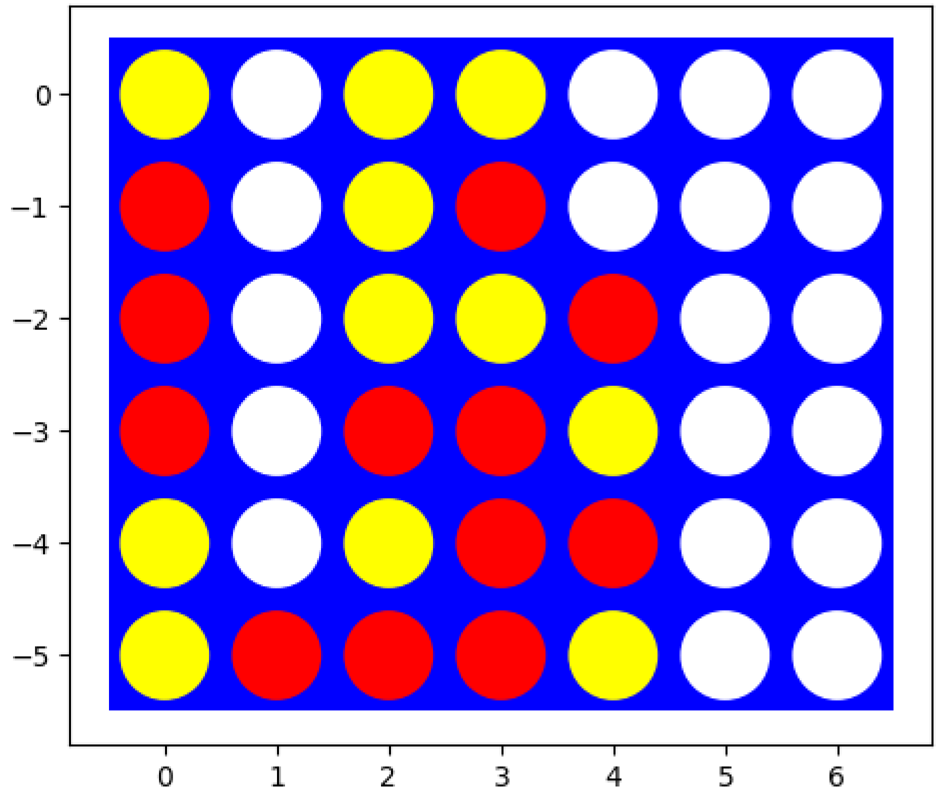
```
[0, 1, 2, 3, 4, 5, 6]
```

In [ ]:

```
# Your code/ answer goes here.
from IPython.display import clear_output

CUTOFF_DEBUG = False
CUTOFF_COUNT = 0

def other(the_int):
    if the_int == 1:
        return -1
    else:
        return 1

def a_b_cutoff_search(board, cutoff=None, player=1, verbose=False, eval_f
    global CUTOFF_DEBUG, CUTOFF_COUNT
    CUTOFF_COUNT = 0
    CUTOFF_DEBUG = verbose
    board = np.array(board)

    value, move = max_value_ab(board, player, -math.inf, +math.inf, 0, cu

    return {"move": move, "value": value}

def max_value_ab(board, player, alpha, beta, depth, cutoff, eval_func):
    global CUTOFF_DEBUG, CUTOFF_COUNT
    CUTOFF_COUNT += 1

    v, terminal = eval_func(board, player)
    if ((cutoff is not None and depth >= cutoff) or terminal):
        if terminal:
            alpha, beta = v, v
        return v, None
    v, move = -math.inf, None

    actions = []
    try:
        actions = ConnectHelp.get_valid_moves(board)
        random.shuffle(actions)
    except:
        actions = ConnectHelp.get_valid_moves(board)

    actions = order_actions_on_util(actions, eval_func, player, board)
    for action in actions:
        v2, a2 = min_value_ab(ConnectHelp.place(action, board, player=pla
        if (v2 > v):
            v, move = v2, action
            alpha = max(alpha, v)
        if v >= beta:
            return v, move
    return v, move

def min_value_ab(board, player, alpha, beta, depth, cutoff, eval_func):
    global CUTOFF_COUNT, CUTOFF_DEBUG
    v, terminal = eval_func(board, player)

    if ((cutoff is not None and depth >= cutoff) or terminal):
        if terminal:
            alpha, beta = v, v
        return v, None
    v, move = +math.inf, None

    actions = []
    try:
        actions = ConnectHelp.get_valid_moves(board)
        random.shuffle(actions)
    except:
        actions = ConnectHelp.get_valid_moves(board)
    actions = order_actions_on_util(actions, eval_func, player, board)
    for action in actions:
        v2, a2 = max_value_ab(ConnectHelp.place(action, board, player=oth
        if v2 < v:
            v, move = v2, action

            beta = min(beta, v)
        if v <= alpha:
            return v, move
    return v, move

def truly_dynamic_environment(players, size=(4, 4), visual=False, board=N
    result = {}
    if board is None:
        board = empty_board(shape=size)
    turn_num = 0
    result['algo_info'] = {
        players[0]['algo'].__name__: {'time': []},
        players[1]['algo'].__name__: {'time': []}
    }
    result['algo_info']
    past_boards = []
    while (ConnectHelp.check_win(board) == 0):  # non-terminal state
        player_turn = turn_num % 2

        start = timer()
        choice = players[player_turn]['algo'](board, **players[player_tur
        end = timer()
        board = ConnectHelp.place(choice, board, player=players[player_tu
        if visual:
            print(f"Utility for {players[player_turn]['algo'].__name__}:
            visualize(board)
            clear_output(wait=True)
        result['algo_info'][players[player_turn]['algo'].__name__]['time'
        past_boards.append(board)
        turn_num += 1
    result['winner'] = ConnectHelp.check_win(board)
    result['turns_taken'] = turn_num
    return result, board, past_boards
```

Code from agents playing each other from Hahsler:

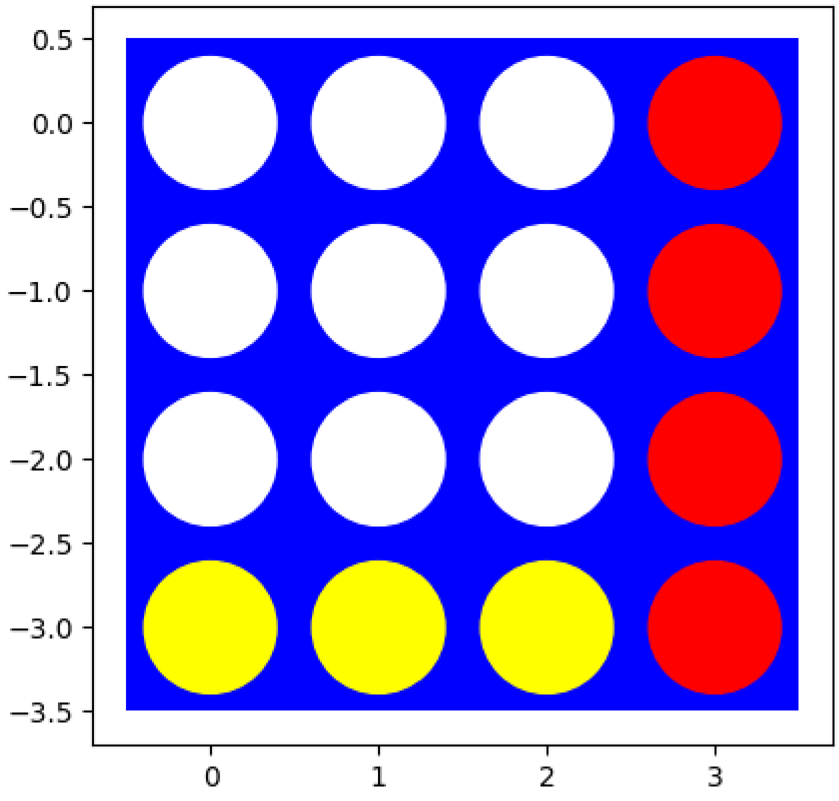
```
In [ ]: playerz = [
            {
                "algo": a_b_cutoff_search,
                "player": 1,
                "args": {
                    'cutoff': 5,
                    'verbose': False
                }
            },
            {
                "algo": random_player,
                "player": -1,
                "args": {}
            }
        ]

        board = [
            [-1, 0, -1, -1, 0, 0, 0],
            [1, 0, -1, 1, 0, 0, 0, ],
            [1, 0, -1, -1, 1, 0, 0, ],
            [1, 0, 1, 1, -1, 0, 0],
            [-1, 0, -1, 1, 1, 0, 0],
            [-1, 1, 1, 1, -1, 0, 0]
        ]

        visualize(board)

        print(a_b_cutoff_search(board, cutoff=6, player=-1, verbose=True))
```

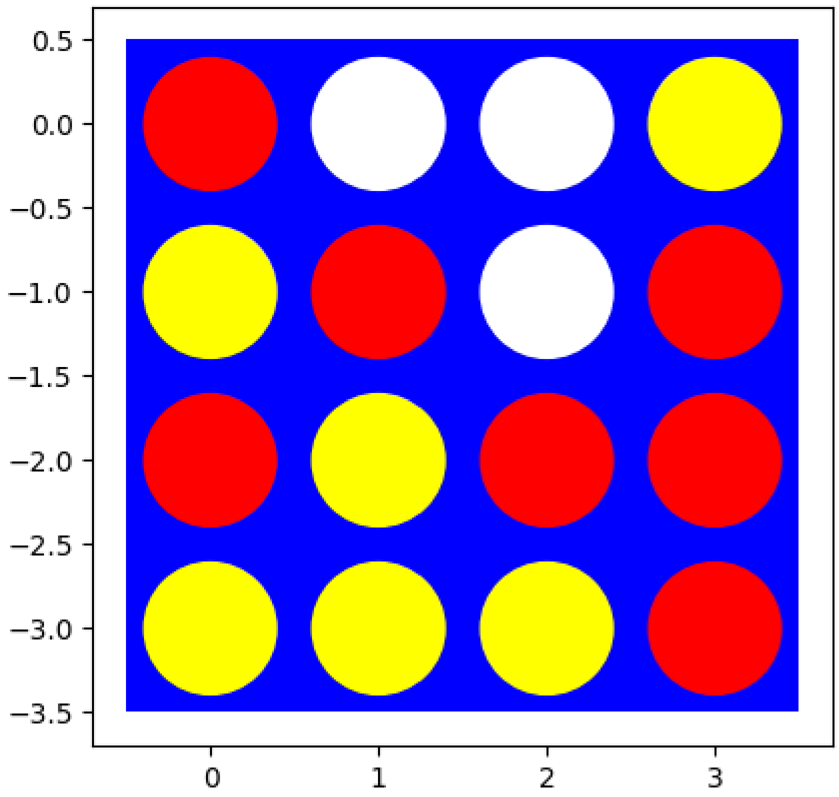


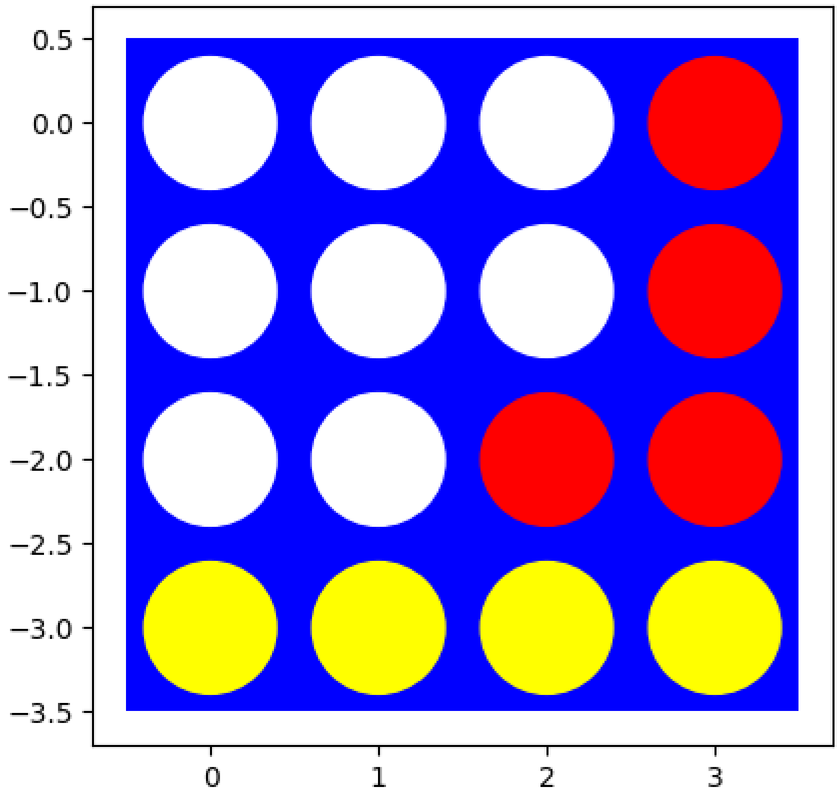
```
{'move': 1, 'value': -2}
```

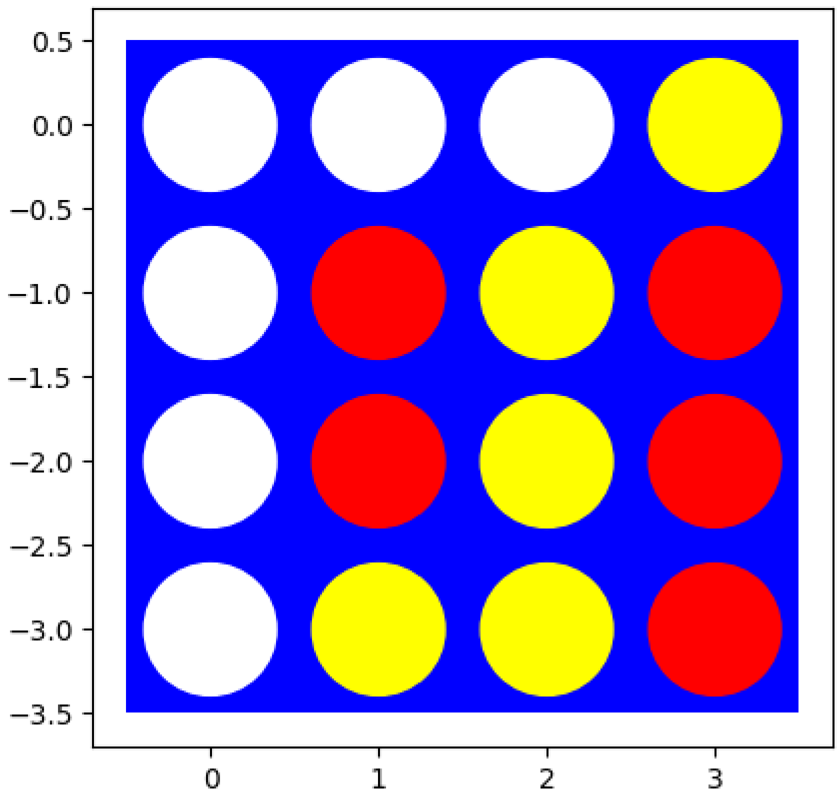
Experiment with the same manually created boards as above to check if the agent spots wining opportunities.

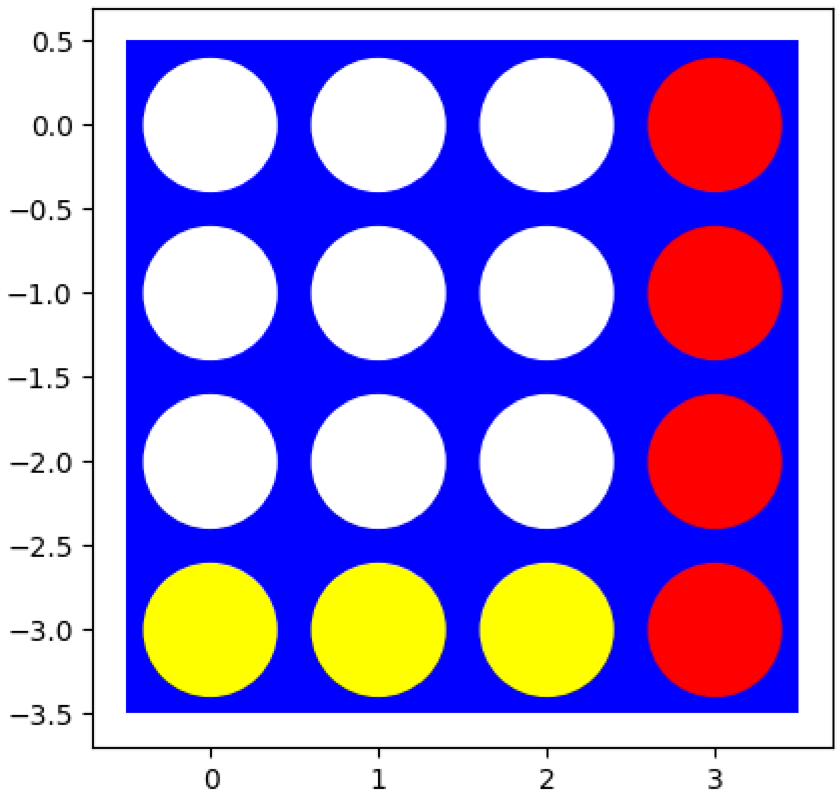
```
In [ ]: # Your code/ answer goes here.
        board1 = [[0,0,0,0],
                  [0,0,0,1],
                  [0,0,0,1],
                  [-1,-1,-1,1]]
        board2 = [[0,0,0,-1],
                  [-1,1,0,1],
                  [1,-1,1,1],
                  [-1,-1,-1,1]]
        board3 = [[0,0,0,1],
                  [0,0,0,1],
                  [0,0,1,1],
                  [0,-1,-1,-1]]
        board4 = [[0,0,0,-1],
                  [0,1,0,1],
                  [0,1,-1,1],
                  [0,-1,-1,1]]
        board5 = [[0,0,0,0],
                  [0,0,0,1],
                  [0,0,0,1],
                  [-1,-1,-1,1]]
        boards = [board1,board2,board3,board4,board5]

        for board in boards:
            visualize(ConnectHelp.place(a_b_cutoff_search(board)['move'],board))
```




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

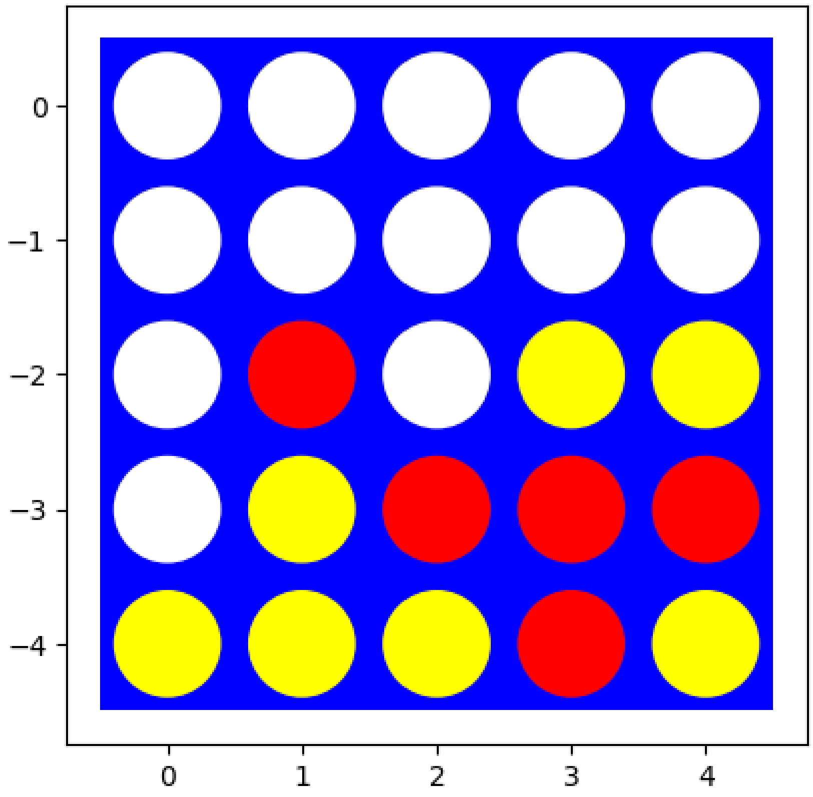




How long does it take to make a move? Start with a smaller board with 4 columns and make the board larger by adding columns.

It took about 2 seconds for all of the boards, which is much faster than the minimax searching without the pruning. Adding another column is as follows:

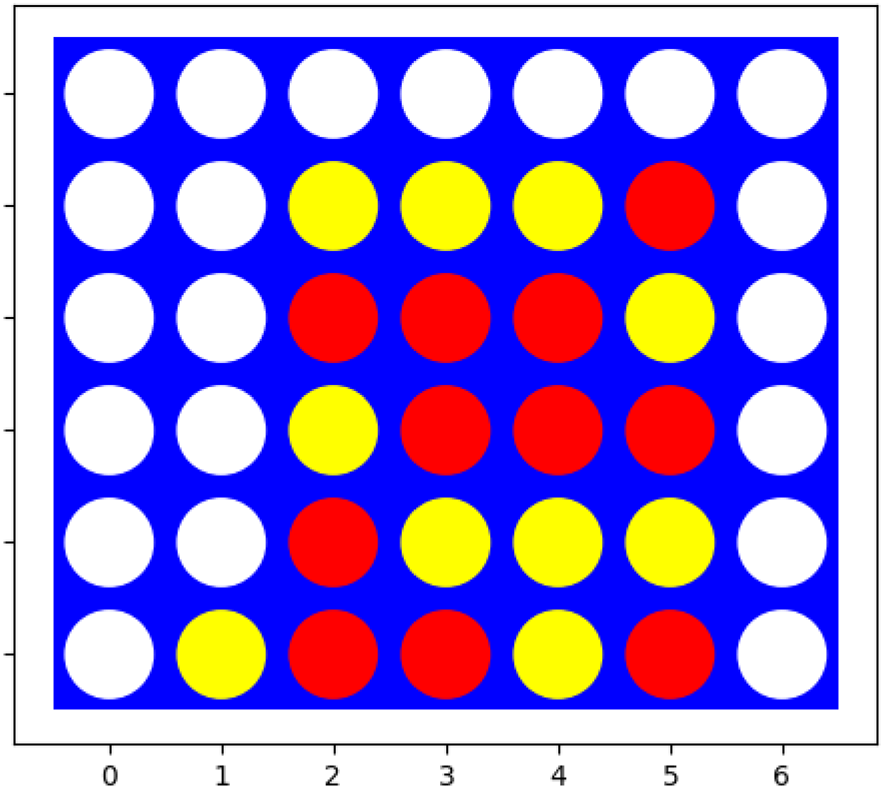
```
In [ ]: board = [[0,0,0,0,0],
                 [0,0,0,0,0],
                   [0,0,0,-1,-1],
                   [0,-1,1,1,1],
                   [-1,-1,-1,1,-1]]

         visualize(ConnectHelp.place(a_b_cutoff_search(board)['move'],board))
```



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 [ ]: # Your code/ answer goes here.
playerz = [
    {
        "algo":a_b_cutoff_search,
        "player":1,
        "args":{
            'cutoff':8,
            'verbose':False
        }
    },
    {
        "algo":a_b_cutoff_search,
        "player":-1,
        "args":{
            'cutoff':3,
            'verbose':False
        }
    }
]

result,final_board,all_boards = truly_dynamic_environment(playerz,size=(6
visualize(final_board)
```



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

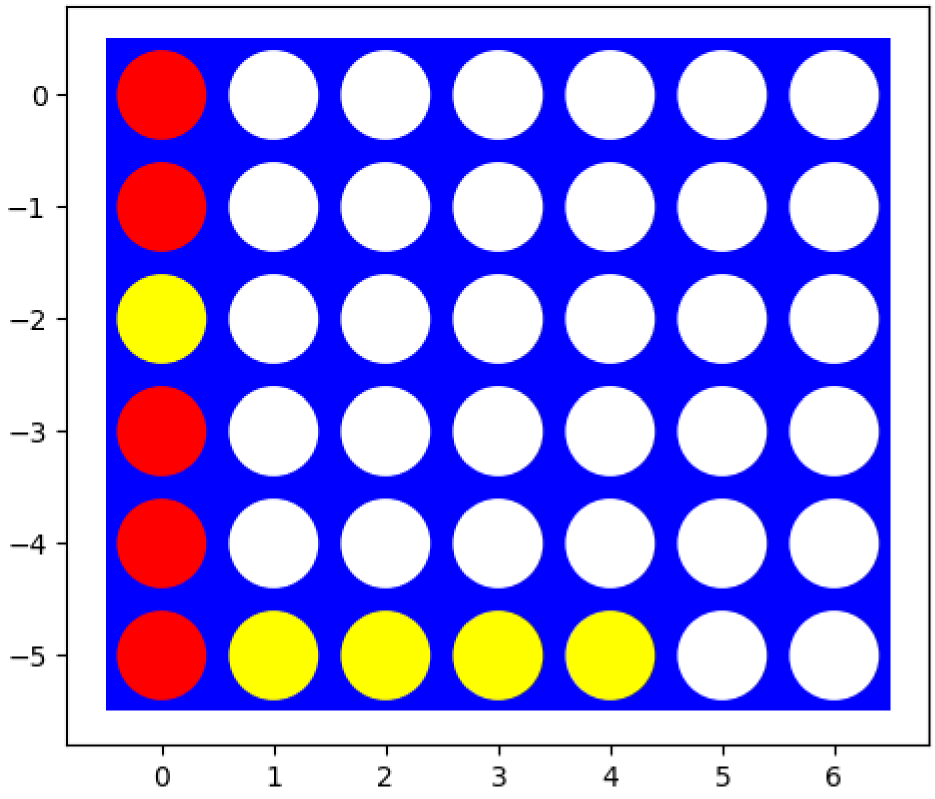
Find another student and let your best agent play against the other student's best player. We will set up a class tournament on Canvas. This tournament will continue after the submission deadline.

I want to be able to play my bot. So here's some code so I can play against the bot.

```
In [ ]: def play_game():
            board = np.zeros((6, 7), dtype=int)  # Create an empty board
            Helper = ConnectHelp()

            visualize(board)

            while True:
                while True:
                    try:
                        user_move = int(input("Enter your move (0-6): "))
                        if user_move < 0 or user_move > 6:
                            raise ValueError("Invalid move. Enter a number betwee
                        if board[0][user_move] != 0:
                            raise ValueError("Column is full. Choose another colu
                        break
                    except ValueError as ve:
                        print(ve)
```

```
        # Update the board with the user's move
        for row in reversed(board):
            if row[user_move] == 0:
                row[user_move] = 1
                break

        clear_output()
        visualize(board)

        # Check for user's win or draw
        win_status = Helper.check_win(board)
        if win_status == 1:
            print("You won!")
            break
        elif win_status == -1:
            print("You lost ding dong")
            break
        elif win_status is None:
            print("Tie")
            break

        # Agent's move (Replace this logic with the agent's move)
        agent_move = a_b_cutoff_search(board, cutoff=5, player=-1, verbos

        # Update the board with the agent's move
        for row in reversed(board):
            if row[agent_move] == 0:
                row[agent_move] = -1  # Agent represented by -1
                break

        clear_output()
        visualize(board)

        # Check for agent's win or draw
        win_status = Helper.check_win(board)
        if win_status == 1:
            print("You won!")
            break
        elif win_status == -1:
            print("You lost ding dong!")
            break
        elif win_status is None:
            print("Tie!")
            break
```

In [ ]:
```
# Start the game

play_game()
```



```
You lost ding dong!
```

Playing against an alpha-beta pruning bot with a cutoff of 5 is really hard to beat as a human. I've had all my friends play it, we all lost. It took five of us playing at once to force a draw.

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

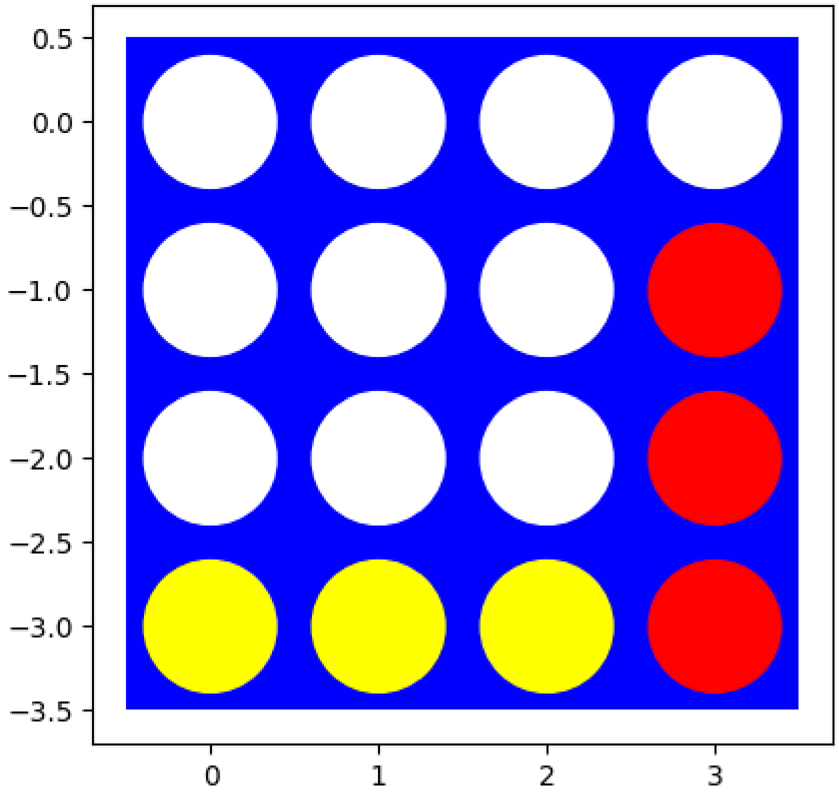
Implement Pure Monte Carlo Search and investigate how this search performs on the test boards that you have used above.

Undergrad attempt, we'll see how it works. Not gonna enter this bot in the competition but we'll see.

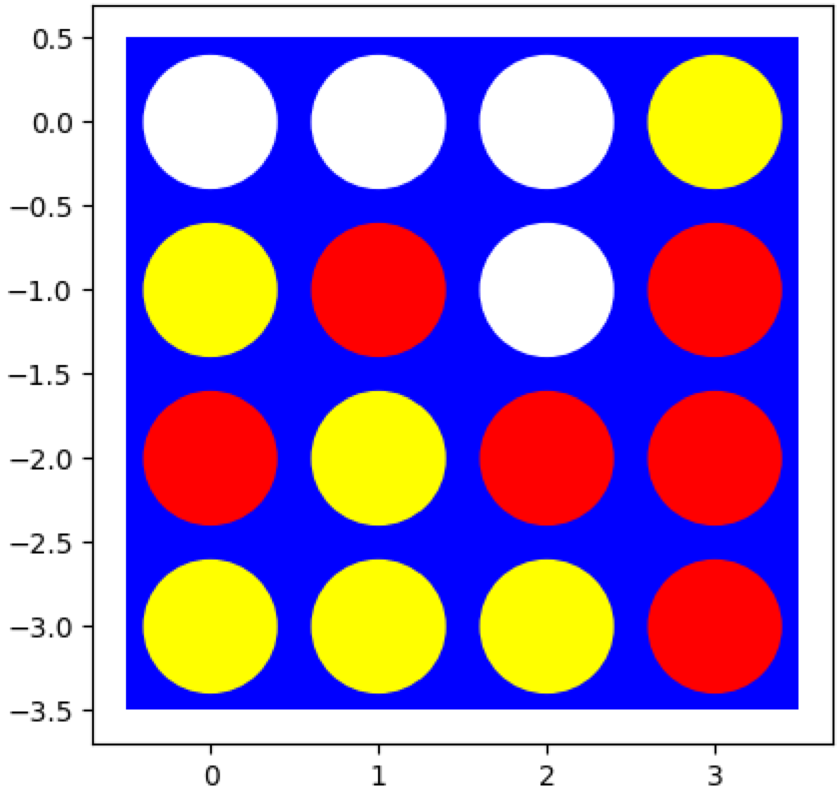
```
In [ ]: # Your code/ answer goes here.
        import random
        import numpy as np
        from scipy.special import softmax

        def heuristic_random_player(state, player=1, verbose=False):
            actions = ConnectHelp.get_valid_moves(state)
            move_values = []

            for move in actions:
```

```
        tmp_state = ConnectHelp.place(move, state, player=player)
        util_val = ConnectHelp.evaluate_board(tmp_state, player)[0]

        if util_val == 1:
            return {"move": move}

        move_values.append(util_val)

    if verbose:
        print(f"""
        Move Values: {move_values}
        Normalized Values: {softmax(move_values)}
        Sum of Normalized Values: {np.sum(softmax(move_values))}
        """)

    softmax_probs = softmax(move_values)
    chosen_move = random.choices(actions, weights=list(softmax_probs))[0]
    return {"move": chosen_move}

def playout(state, action, player=1, cutoff_val=3, playout_func=random_pl
    state = ConnectHelp.place(action, state, player=player)
    current_player = other(player)

    while True:
        try:
            return ConnectHelp.calc_utility(player, state)
        except:
            action = playout_func(state, player=current_player)['move']
            state = ConnectHelp.place(action, state, player=current_playe
            current_player = other(current_player)



#from way above
for board in boards:
    print(ConnectHelp.evaluate_board(board, player=1))  # Evaluating the
    visualize(board)  # Visualizing the board
    print(heuristic_random_player(board, verbose=True))  # Executing heur
```

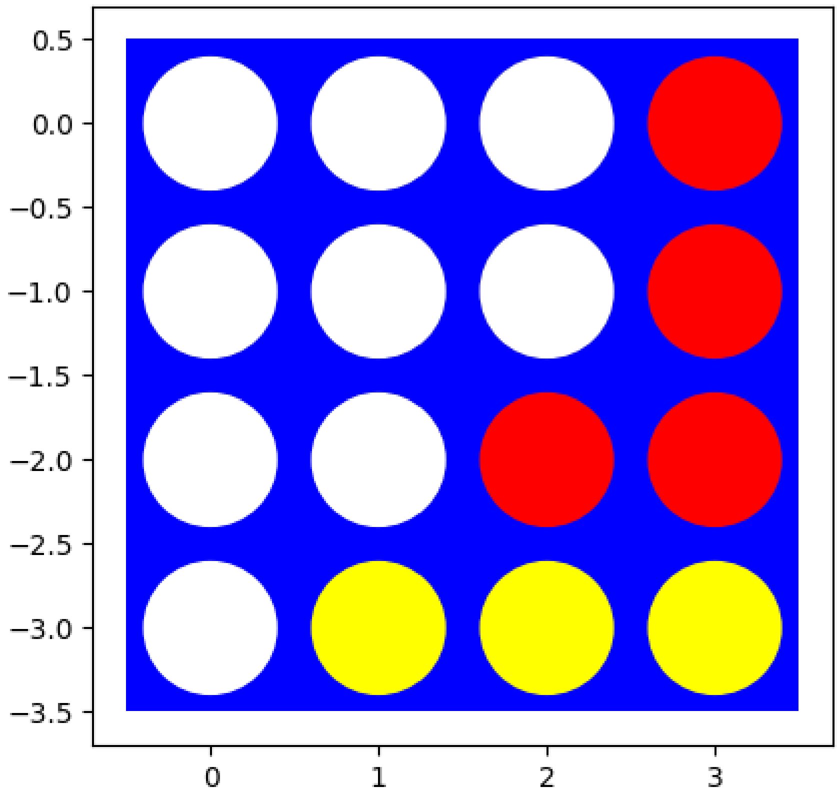
```
(0.12435300177159614, False)
```



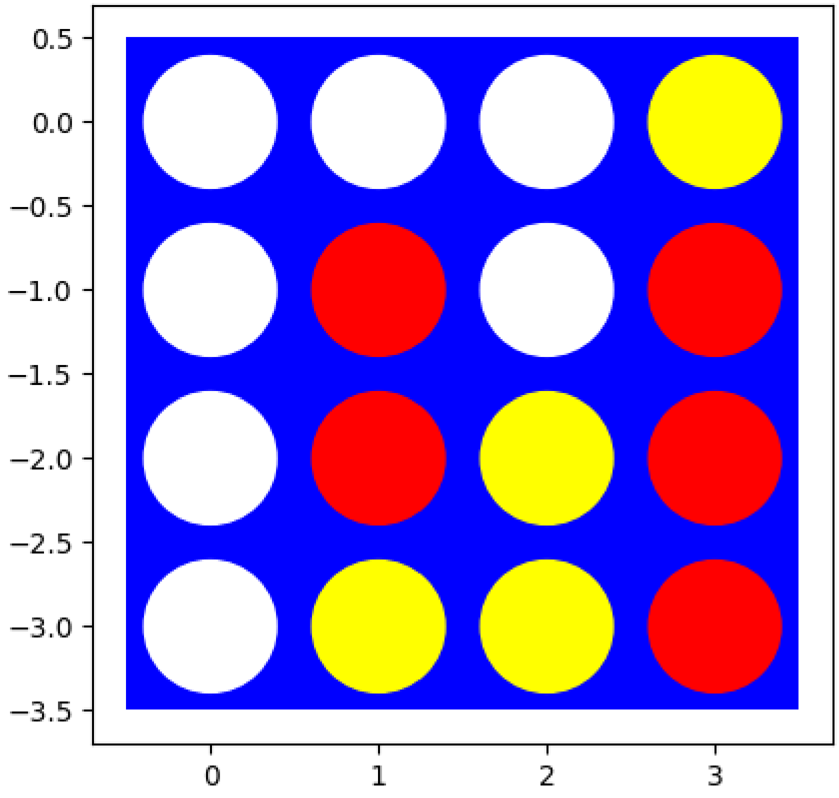
```
        Move Values: [0.2449186624037092, 0.2449186624037092, 0.35835739
835078595, 2]
        Normalized Values: [0.11230855 0.11230855 0.12579942 0.64958347]
        Sum of Normalized Values: 1.0

{'move': 0}
(0.12435300177159614, False)
```



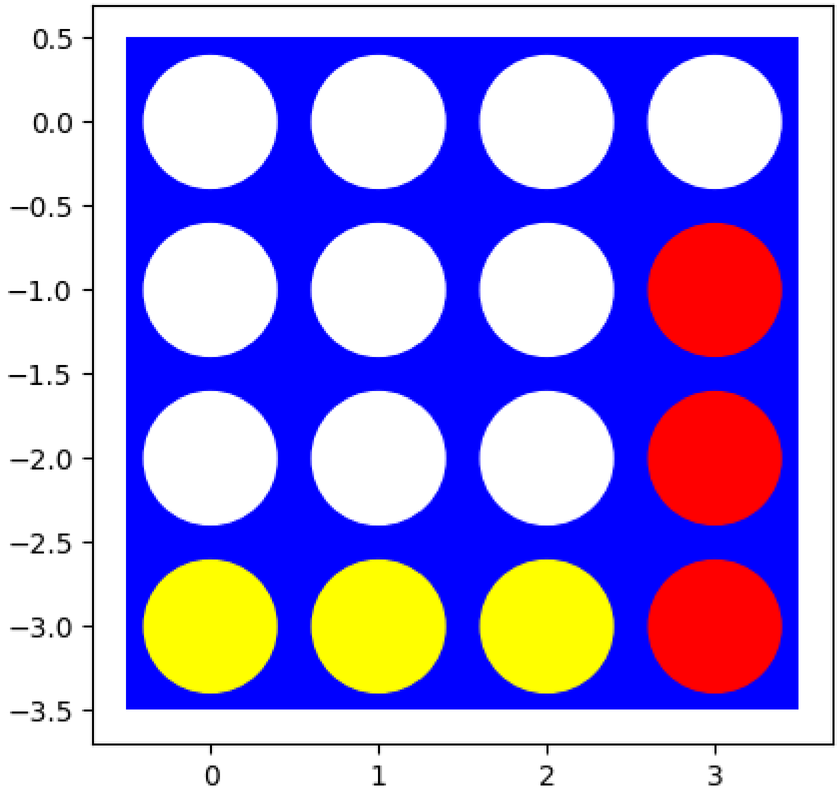
```
        Move Values: [2, 0.12435300177159614, 0.6351489523872873]
        Normalized Values: [0.7098872  0.10879432 0.18131848]
        Sum of Normalized Values: 1.0000000000000002

{'move': 1}
(0.0, False)
```



```
        Move Values: [0.2449186624037092, 0.2449186624037092, 0.24491866
24037092]
        Normalized Values: [0.33333333 0.33333333 0.33333333]
        Sum of Normalized Values: 1.0

{'move': 2}
(0.12435300177159614, False)
```



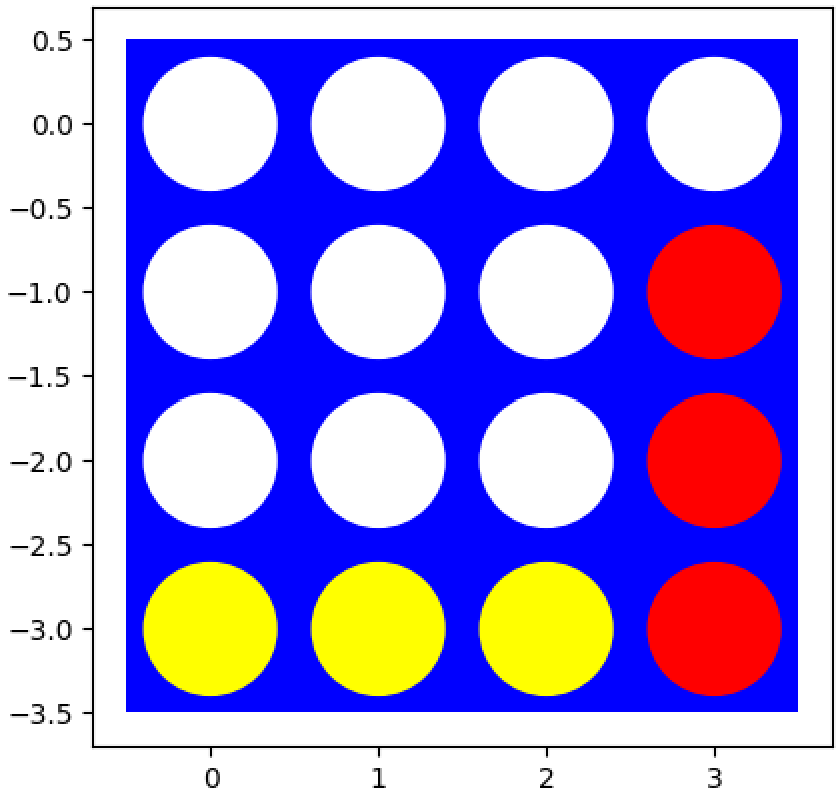
```
        Move Values: [0.12435300177159614, 0.2449186624037092, 0.3583573
9835078595]
        Normalized Values: [0.29483002 0.33260801 0.37256197]
        Sum of Normalized Values: 1.0

{'move': 0}
(0.12435300177159614, False)
```



```
        Move Values: [0.2449186624037092, 0.2449186624037092, 0.35835739
835078595, 2]
        Normalized Values: [0.11230855 0.11230855 0.12579942 0.64958347]
        Sum of Normalized Values: 1.0

{'move': 1}
```

In [ ]:

```python
def playouts(board, action, player=1, N=100, playout_func=random_player):
    playout_results = np.array([playout(board, action, player, cutoff_val
    return playout_results


visualize(board)

u = playouts(board, 0)
print("Playout results:", u)

mean_utility = np.mean(u)
print(f"mean utility: {mean_utility}")

# Calculate win, loss, and draw probabilities using NumPy functions
p_win = np.sum(u == 1) / len(u)
p_loss = np.sum(u == -1) / len(u)
p_draw = np.sum(u == 0) / len(u)

print(f"win probability: {p_win}")
print(f"loss probability: {p_loss}")
print(f"draw probability: {p_draw}")
```



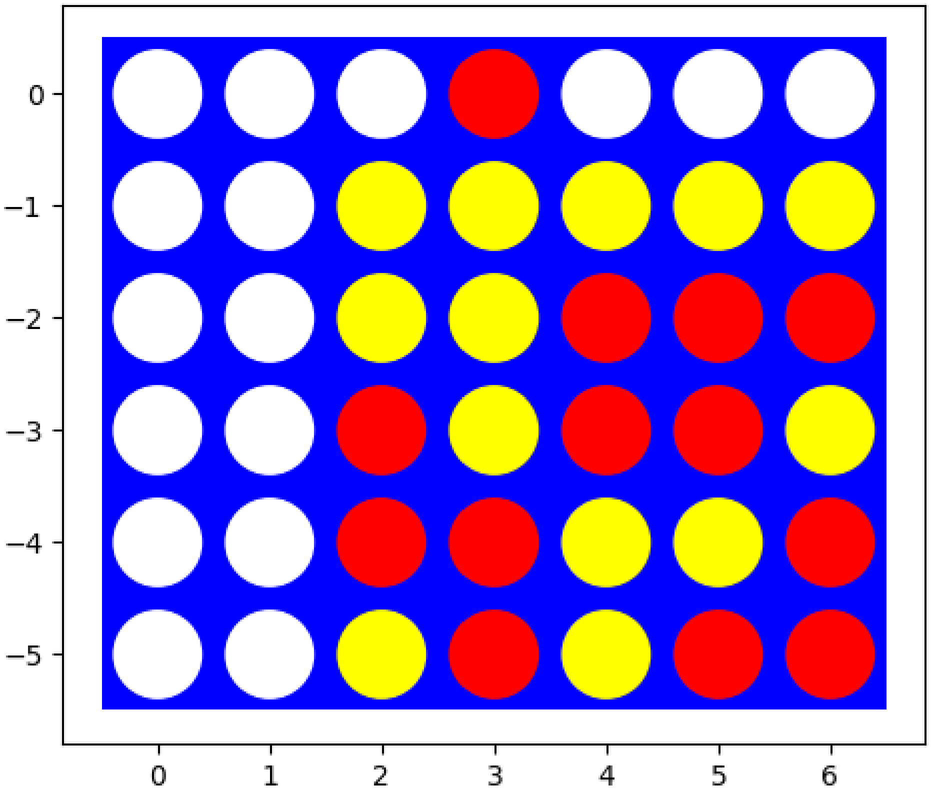
```
Playout results: [ 1 -1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
1  1 -1  1  1  1
  1  1  1  1  1  1  1  1  1  1  0  1  1  1  0  1  1  0  1  1  1  1  0  1
  1  0  1 -1  1  1  1  1  1  1  0  1  0  1  0  1  1  1  1  0  1  1  0  1
  0  1  1  0  1 -1  1  1  1  1  0  1  1  1  1  0  1  1  1  1  1  1  0 -1
  1  1  1  1]
mean utility: 0.75
win probability: 0.8
loss probability: 0.05
draw probability: 0.15
```

In [ ]:
```python
def pmcs(board, num_playouts=50, player=1, verbose=False, playout_func=ra
    debug_mode = verbose
    available_moves = ConnectHelp.get_valid_moves(board)
    playouts_per_action = num_playouts // len(available_moves)

    if debug_mode:
        print(f"Available Moves: {available_moves} ({playouts_per_action}

    mean_utilities = {
        move: np.mean(playouts(board, move, player, N=playouts_per_action
        for move in available_moves
    }

    if debug_mode:
        print(mean_utilities)

    selected_action = max(mean_utilities, key=mean_utilities.get)
    return {"move": selected_action}
```

Also code from Hahsler about playing agents against each other.

In [ ]:
```python
playerz = [
    {
        "algo":pmcs,
        "player":1,
        "args":{
            'num_playouts':3000,
            'verbose':False,
            'player':1
        }
    },
    {
        "algo":a_b_cutoff_search,
        "player":-1,
        "args":{
            'verbose':False,
            'cutoff':8,
            #this is a smart bot yikes, we'll see how it goes
            'player':-1
        }
    }
]

result,final_board,past_boards = truly_dynamic_environment(playerz,size=(
```

```
Utility for a_b_cutoff_search: (-2, True)
```

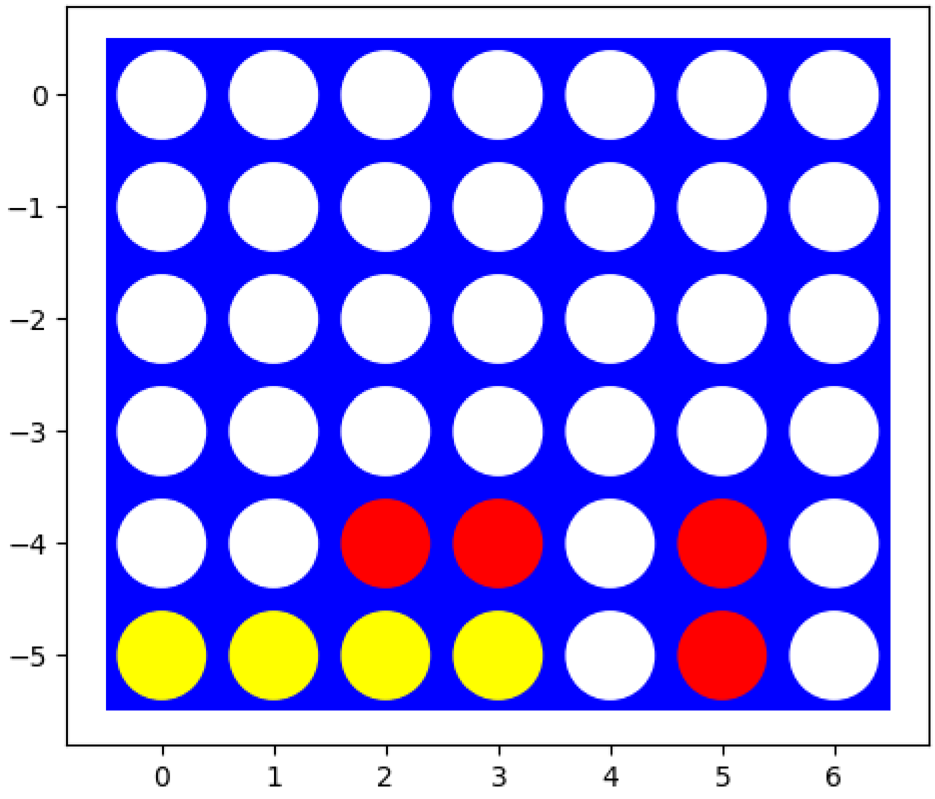


So in an alpha-beta bot with cutoff of 8, pure Monte Carlo search loses most of the time. It took 3.5 minutes, so I'm not gonna rerun it more than once.

Testing Different Cutoffs for Time/Victories

In [ ]:
```
playerz = [
    {
        "algo":a_b_cutoff_search,
        "player":1,
        "args":{
            'cutoff':1,
            'verbose':False,
            'player':1
        }
    },
    {
        "algo":a_b_cutoff_search,
        "player":-1,
        "args":{
            'verbose':False,
            'cutoff':7,
            #this is a smart bot yikes, we'll see how it goes
            'player':-1
        }
    }
]

result,final_board,past_boards = truly_dynamic_environment(playerz,size=(
```

```
Utility for a_b_cutoff_search: (-2, True)
```



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.
#i have no idea
```