Adversarial Search: Playing Connect 4

Instructions

Total Points: Undegraduates 10, graduate students 11

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 [1 point]

Define the components of the search problem:

- Initial state
- Actions
- Transition model
- Goal state

In [1]:
```
# Your code/answer goes here.
#At initial both Players are at the same state then the user is given the first move to choose a column from.
#the action is taken as per the column and for the termination states test whenever the diagnoas or a row or a
#check_Winner Function the game is being terminated
```

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

In [2]:
```
# Your code/ answer goes here.
3**16 #For 4x4 board
```

Out[2]:
```
43046721
```

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

In [3]:
```
# Your code/ answer goes here.
#The minmax tree can be either of 1 evel or of two levels depending upon the mode you go for. In 1 it is only
#While in hard Mode it is concerned with it's as well as the other human player's moves.
```

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

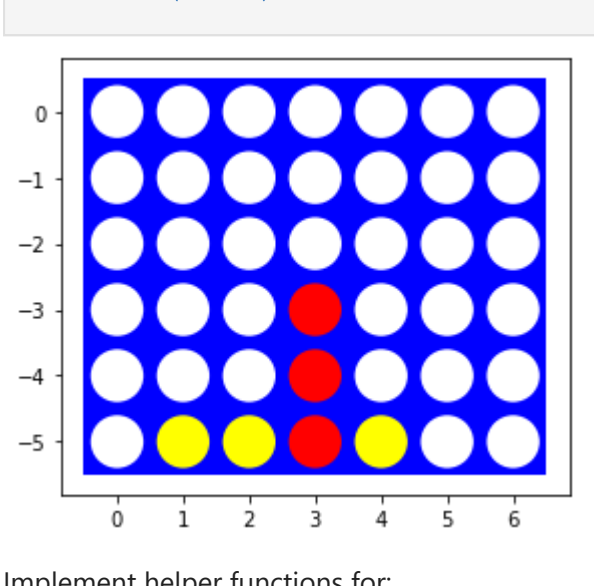
Use a numpy character array as the board.

In [4]:
```
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]]
```

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 and player is the player whose next move it is and who the agent should play.

In [5]:
```
# 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:

- The transition model (result).
- The utility function.
- Check for terminal states.
- A check for available actions.

Make sure that all these functions work with boards of different sizes.

Implement an agent that plays randomly and let two random agents play against each other 1000 times. How often does each player win? Is the result expected?

In [5]:
```
# Your code/ answer goes here.
import random
import numpy as np
import math
import pygame

BLUE = (0, 0, 255)  #RGB for blue
BLACK = (0, 0, 0)   #RGB for black
RED = (255, 0, 0)   #RGB for red
YELLOW = (255, 255, 0)#RGB for yellow

rows = 4
cols = 4

def valid_move_check(board, col):
    return board[rows - 1][col] == 0


def get_free_row(board, col):  # tells the lowest available row in a column
    for i in range(rows):
        if board[i][col] == 0:
            return i

def ball_drop(board, row, col, ball):
    board[row][col] = ball

def initialize_board():
    board = np.zeros((rows, cols))
    return board

def print_board(board):
    print(np.flip(board, 0)) #flips board so 0 row starts from bottom
```

```
pygame 2.1.0 (SDL 2.0.16, Python 3.8.8)
Hello from the pygame community. https://www.pygame.org/contribute.html
```

Task 3: Minimax Search with Alpha-Beta Pruning [4 points]

Implement the search starting from a given board and specifying the player.

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

In [6]:
```
# Your code/ answer goes here.
def check_winner(board, ball):
    for i in range(cols - 3):     # Check rows for winner
        for j in range(rows):
            if board[j][i] == ball and board[j][i + 1] == ball and board[j][i + 2] == ball and board[j][
                i + 3] == ball:
                return True

    for i in range(cols):         # Check columns for winner
        for j in range(rows - 3):
            if board[j][i] == ball and board[j + 1][i] == ball and board[j + 2][i] == ball and board[j + 3][
                i] == ball:
                return True

    for i in range(cols - 3):     # Check diagonals for winner with upper side at left
        for j in range(rows - 3):
            if board[j][i] == ball and board[j + 1][i + 1] == ball and board[j + 2][i + 2] == ball and board[j
                i + 3] == ball:
                return True

    for i in range(cols - 3):     # Check diagonals for winner with upper side at right
        for j in range(3, rows):
            if board[j][i] == ball and board[j - 1][i + 1] == ball and board[j - 2][i + 2] == ball and board[j
                i + 3] == ball:
                return True
```

In [7]:
```
def make_board(board):   #GUI
    for c in range(cols):
        for r in range(rows):
            pygame.draw.rect(screen, BLUE, (c * ballSize, r * ballSize + ballSize, ballSize, ballSize))
            pygame.draw.circle(screen, BLACK, (
            int(c * ballSize + ballSize / 2), int(r * ballSize + ballSize + ballSize / 2)), radius)

    for c in range(cols):
        for r in range(rows):
            if board[r][c] == 1:
                pygame.draw.circle(screen, RED, (
                int(c * ballSize + ballSize / 2), height - int(r * ballSize + ballSize / 2)), radius)
            elif board[r][c] == 2:
                pygame.draw.circle(screen, YELLOW, (
                int(c * ballSize + ballSize / 2), height - int(r * ballSize + ballSize / 2)), radius)
    pygame.display.update()

def evaluation(board,ball):   #we evaluate the board for the future move

    priority=0

    arr_mid=[int(i) for i in list(board[:,cols//2])]
    mid_priority=arr_mid.count(ball)
    priority=priority+(mid_priority*3)

    for i in range(rows):#Horizontal evaluation
        arr_row=[int(j) for j in list(board[i,:])] #gets rows one by one   int(i) is syntax and check so value
        for k in range(cols -3): #so that we stop when last 3 places are left in row cz we cant make any 4 grou
            cutting=arr_row[k:k+4]#gives values in group of 4
            if cutting.count(ball) == 4:
                priority += 100
            elif cutting.count(ball) == 3 and cutting.count(0) == 1: # if we find a possible combination where
                priority+=5
            elif cutting.count(ball) == 2 and cutting.count(0) == 2:
                priority+=2
            if cutting.count(1) == 3 and cutting.count(0) == 1:
                priority-=4
    for i in range(rows-3):  #Diagonal evaluation with upper at right
        for j in range(cols-3):
            cutting=[board[i+k][j+k] for k in range(4)]
            if cutting.count(ball) == 4:
                priority += 100
            elif cutting.count(ball) == 3 and cutting.count(0) == 1:  # if we find a possible combination where
                priority += 5
            elif cutting.count(ball) == 2 and cutting.count(0) == 2:
                priority += 2
            if cutting.count(1) == 3 and cutting.count(0) == 1:
                priority -= 4

    for i in range(rows-3):  #Diagonal evaluation
        for j in range(cols-3):
            cutting=[board[i+3-k][j+k] for k in range(4)]
            if cutting.count(ball) == 4:
                priority += 100
            elif cutting.count(ball) == 3 and cutting.count(0) == 1:  # if we find a possible combination where
                priority += 5
            elif cutting.count(ball) == 2 and cutting.count(0) == 2:
                priority += 2
            if cutting.count(1) == 3 and cutting.count(0) == 1:
                priority -= 4

    return priority

def find_all_valid_moves(board):  #check in which row we can drop ball
    valid_moves=[]
    for i in range(cols):
        if valid_move_check(board,i):
            valid_moves.append(i)#we add valid column number
    return valid_moves

def is_terminal_node(board): #check if anyone has won or board is filled
    return len(find_all_valid_moves(board))==0 or check_winner(board,1) or check_winner(board,2)
def minmax(board,maxplayer,alpha,beta,depth):
    terminal_node=is_terminal_node(board) #checks if we reached terminal node
    if depth==0 or terminal_node:
        if terminal_node:
            if check_winner(board,2): #2 represents AI ball
                return (100000000000000,None)  #first value is score and second is column number
            elif check_winner(board,1):
                return (-100000000000000,None)
            else:                          #if board is full
                return (0,None)
        elif depth==0:
            return (evaluation(board,2),None)

    elif maxplayer: #if we are at miximizng level
        valid_moves=find_all_valid_moves(board)
        priority=-math.inf #negative infinity
        prior_col=random.choice(valid_moves)
        for poss_col in valid_moves:
            new_board = board.copy()
            poss_row=get_free_row(new_board,poss_col) #gets the row number of that column
            ball_drop(new_board,poss_row,poss_col,2)   #check by dropping AI ball
            updated_prior=minmax(new_board,False,alpha,beta,depth-1)[0] #0th index represnets the score returne
            if(priority<updated_prior):
                priority=updated_prior
                prior_col=poss_col
            alpha=max(alpha,priority) #alpha beta pruning
            if alpha>=beta:
                break

        return priority,prior_col

    else: #if we are at minimizng level
        valid_moves = find_all_valid_moves(board)
        prior_col=random.choice(valid_moves)
        priority = math.inf
        for poss_col in valid_moves:
            new_board = board.copy()
            poss_row = get_free_row(new_board, poss_col)
            ball_drop(new_board, poss_row, poss_col, 1) #check by dropping human ball
            updated_prior = minmax(new_board, True,alpha,beta, depth - 1)[0] #we collect the priority score not
            if (priority>updated_prior):
                prior_col = poss_col
                priority=updated_prior
            beta = min(beta, priority)
            if alpha >= beta:
                break

        return priority,prior_col
```

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

In [10]:
```
# Your code/ answer goes here.
#In easy mode the agent does not spot winning opportunities but in hard mode the agent spots the winning opport
```

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 [11]:
```
# Your code/ answer goes here.
#Just a narrow of a second
```

Move ordering

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 [12]:
```
# Your code/ answer goes here.

#The minmax tree can be either of 1 level or of two levels depending upon the mode you go for. In easy mode it
#While in hard Mode it is concerned with it's as well as the other human player's moves.
```

The first few moves

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

In [12]:
```
# Your code/ answer goes here.
#If easy mode then the first move can be of anywhere but in the hard mode since evaluating
#up to depth of three levels so most likely in the midddle since in this way it makes the opponent play to have
```

Playtime

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

In [13]:
```
# Your code/ answer goes here.
game_over_check = False
turn = 0 #human turn is first
difficulty=0
print("Select Difficulty \n")
print("1) Easy \n")
print("2) Hard \n")
difficulty=input('Choice : ')
difficulty=int(difficulty)

while difficulty !=1 and difficulty!=2:
    print("Select Difficulty \n")
    print("1) Easy \n")
    print("2) Hard \n")
    difficulty = input('Choice : ')
    difficulty = int(difficulty)

if difficulty==2: #hard is level 2 and easy is 1
    difficulty=3

board = initialize_board()
print_board(board)
pygame.init()

ballSize = 85
width = cols * ballSize
height = (rows + 1) * ballSize
size = (width, height)
radius = int(ballSize / 2 - 5)

screen = pygame.display.set_mode(size)
make_board(board)
pygame.display.update()
myfont = pygame.font.SysFont("Helvetica", 65)



while not game_over_check:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            exit()

        if event.type == pygame.MOUSEMOTION:
            pygame.draw.rect(screen, BLACK, (0, 0, width, ballSize))
            ballPos = event.pos[0]
            if turn == 0: # red for human
                pygame.draw.circle(screen, RED, (ballPos, int(ballSize / 2)), radius)
        pygame.display.update()

        if event.type == pygame.MOUSEBUTTONDOWN:
            pygame.draw.rect(screen, BLACK, (0, 0, width, ballSize))
            if turn == 0:
                ballPos = event.pos[0]
                col = int(math.floor(ballPos / ballSize))

                if valid_move_check(board, col):
                    row = get_free_row(board, col)
                    ball_drop(board, row, col, 1)

                    if check_winner(board, 1):
                        label = myfont.render("YOU WIN CONGRATS", 1, RED)
                        screen.blit(label, (40, 10))
                        game_over_check = True
                    turn = (turn + 1) % 2
                    print_board(board)
                    make_board(board)

    # AI turn
    if turn==1 and not game_over_check:
        func_score,best_col=minmax(board,True,-math.inf,math.inf,difficulty) #minmax function calling

        if valid_move_check(board, best_col):
            row = get_free_row(board, best_col)
            ball_drop(board, row, best_col, 2)

            if check_winner(board, 2):
                label = myfont.render("A.I WINS", 1, YELLOW)
                screen.blit(label, (40, 10))
                game_over_check = True

            print_board(board)
            make_board(board)

            turn = (turn+1) % 2

    if game_over_check:
        pygame.time.wait(3500) #wait before disappearing screen
```

```
Select Difficulty

1) Easy

2) Hard

Choice : 2
[[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. 1. 0.]]
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 2. 0.]
 [0. 0. 1. 0.]]
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 2. 0.]
 [0. 1. 1. 0.]]
[[0. 0. 0. 0.]
 [0. 0. 2. 0.]
 [0. 0. 2. 0.]
 [0. 1. 1. 0.]]
[[0. 0. 0. 0.]
 [0. 0. 2. 0.]
 [0. 1. 2. 0.]
 [0. 1. 1. 0.]]
[[0. 0. 0. 0.]
 [0. 0. 2. 0.]
 [0. 1. 2. 0.]
 [2. 1. 1. 0.]]
[[0. 0. 0. 0.]
 [0. 0. 2. 0.]
 [1. 1. 2. 0.]
 [2. 1. 1. 0.]]
[[0. 0. 0. 0.]
 [0. 2. 2. 0.]
 [1. 1. 2. 0.]
 [2. 1. 1. 0.]]
[[0. 0. 0. 0.]
 [0. 2. 2. 0.]
 [1. 1. 2. 0.]
 [2. 1. 1. 1.]]
[[0. 0. 0. 0.]
 [2. 2. 2. 0.]
 [1. 1. 2. 0.]
 [2. 1. 1. 1.]]
[[1. 0. 0. 0.]
 [2. 2. 2. 0.]
 [1. 1. 2. 0.]
 [2. 1. 1. 1.]]
[[1. 0. 2. 0.]
 [2. 2. 2. 0.]
 [1. 1. 2. 0.]
 [2. 1. 1. 1.]]
[[1. 0. 2. 0.]
 [2. 2. 2. 0.]
 [1. 1. 2. 1.]
 [2. 1. 1. 1.]]
[[1. 0. 2. 0.]
 [2. 2. 2. 2.]
 [1. 1. 2. 1.]
 [2. 1. 1. 1.]]
```

Task 4: Heuristic Alpha-Beta Tree Search [3 points]

Heuristic evaluation function

Define and implement a heuristic evaluation function.

In [10]:
```
# Your code/ answer goes here.
def minmax(board,maxplayer,alpha,beta,depth):
    terminal_node=is_terminal_node(board) #checks if we reached terminal node
    if depth==0 or terminal_node:
        if terminal_node:
            if check_winner(board,2): #2 represents AI ball
                return (100000000000000,None)  #first value is score and second is column number
            elif check_winner(board,1):
                return (-100000000000000,None)
            else:                          #if board is full
                return (0,None)
        elif depth==0:
            return (evaluation(board,2),None)

    elif maxplayer: #if we are at miximizng level
        valid_moves=find_all_valid_moves(board)
        priority=-math.inf #negative infinity
        prior_col=random.choice(valid_moves)
        for poss_col in valid_moves:
            new_board = board.copy()
            poss_row=get_free_row(new_board,poss_col) #gets the row number of that column
            ball_drop(new_board,poss_row,poss_col,2)  #check by dropping AI ball
            updated_prior=minmax(new_board,False,alpha,beta,depth-1)[0] #0th index represnets the score returne
            if(priority<updated_prior):
                priority=updated_prior
                prior_col=poss_col
            alpha=max(alpha,priority) #alpha beta pruning
            if alpha>=beta:
                break

        return priority,prior_col

    else: #if we are at minimizng level
        valid_moves = find_all_valid_moves(board)
        prior_col=random.choice(valid_moves)
        priority = math.inf
        for poss_col in valid_moves:
            new_board = board.copy()
            poss_row = get_free_row(new_board, poss_col)
            ball_drop(new_board, poss_row, poss_col, 1) #check by dropping human ball
            updated_prior = minmax(new_board, True,alpha,beta, depth - 1)[0] #we collect the priority score not
            if (priority>updated_prior):
                prior_col = poss_col
                priority=updated_prior
            beta = min(beta, priority)
            if alpha >= beta:
                break

        return priority,prior_col
```

Cutting off search

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 [11]:
```
# Your code/ answer goes here.
#It always evalutes the AI to the best of possible moves i.e., by returning max value for it and min  value for
```

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

In [12]:
```
# Your code/ answer goes here.
#In easy mode yes it spots it but in hard mode no it doesn't
```

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 [13]:
```
# Your code/ answer goes here.
#Just a narrow of a second
```

Playtime

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 [13]:
```
# Your code/ answer goes here.
rows = 7
cols = 6
game_over_check = False
turn = 0 #human turn is first
difficulty=0
print("Select Difficulty \n")
print("1) Easy \n")
print("2) Hard \n")
difficulty=input('Choice : ')
difficulty=int(difficulty)

while difficulty !=1 and difficulty!=2:
    print("Select Difficulty \n")
    print("1) Easy \n")
    print("2) Hard \n")
    difficulty = input('Choice : ')
    difficulty = int(difficulty)

if difficulty==2: #hard is level 3 and easy is 1
    difficulty=3

board = initialize_board()
print_board(board)
pygame.init()

ballSize = 85
width = cols * ballSize
height = (rows + 1) * ballSize
size = (width, height)
radius = int(ballSize / 2 - 5)

screen = pygame.display.set_mode(size)
make_board(board)
pygame.display.update()
myfont = pygame.font.SysFont("Helvetica", 65)



while not game_over_check:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            exit()

        if event.type == pygame.MOUSEMOTION:
            pygame.draw.rect(screen, BLACK, (0, 0, width, ballSize))
            ballPos = event.pos[0]
            if turn == 0: # red for human
                pygame.draw.circle(screen, RED, (ballPos, int(ballSize / 2)), radius)
        pygame.display.update()

        if event.type == pygame.MOUSEBUTTONDOWN:
            pygame.draw.rect(screen, BLACK, (0, 0, width, ballSize))
            if turn == 0:
                ballPos = event.pos[0]
                col = int(math.floor(ballPos / ballSize))

                if valid_move_check(board, col):
                    row = get_free_row(board, col)
                    ball_drop(board, row, col, 1)

                    if check_winner(board, 1):
                        label = myfont.render("YOU WIN CONGRATS", 1, RED)
                        screen.blit(label, (40, 10))
                        game_over_check = True
                    turn = (turn + 1) % 2
                    print_board(board)
                    make_board(board)

    # AI turn
    if turn==1 and not game_over_check:
        func_score,best_col=minmax(board,True,-math.inf,math.inf,difficulty) #minmax function calling

        if valid_move_check(board, best_col):
            row = get_free_row(board, best_col)
            ball_drop(board, row, best_col, 2)

            if check_winner(board, 2):
                label = myfont.render("AI WINS", 1, YELLOW)
                screen.blit(label, (40, 10))
                game_over_check = True

            print_board(board)
            make_board(board)

            turn = (turn+1) % 2

    if game_over_check:
        pygame.time.wait(3500) #wait before disappearing screen
```

```
Select Difficulty

1) Easy

2) Hard

Choice : 2
[[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.]]
[[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. 1. 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. 1. 2. 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. 1. 0. 0. 0.]
 [0. 0. 1. 2. 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. 1. 2. 0. 0.]
 [0. 0. 1. 2. 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. 1. 0. 0. 0.]
 [0. 0. 1. 2. 0. 0.]
 [0. 0. 1. 2. 0. 0.]]
[[0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 2. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0.]
 [0. 0. 1. 2. 0. 0.]
 [0. 0. 1. 2. 0. 0.]]
[[0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 2. 0. 0. 0.]
 [0. 0. 1. 1. 0. 0.]
 [0. 0. 1. 2. 0. 0.]
 [0. 0. 1. 2. 0. 0.]]
[[0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 2. 2. 0. 0.]
 [0. 0. 1. 1. 0. 0.]
 [0. 0. 1. 2. 0. 0.]
 [0. 0. 1. 2. 0. 0.]]
[[0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 2. 2. 0. 0.]
 [0. 0. 1. 1. 0. 0.]
 [0. 0. 1. 2. 0. 0.]
 [0. 0. 1. 2. 1. 0.]]
[[0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 2. 0. 0. 0.]
 [0. 0. 2. 2. 0. 0.]
 [0. 0. 1. 1. 0. 0.]
 [0. 0. 1. 2. 0. 0.]
 [0. 0. 1. 2. 1. 0.]]
[[0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0.]
 [0. 0. 2. 0. 0. 0.]
 [0. 0. 2. 2. 0. 0.]
 [0. 0. 1. 1. 0. 0.]
 [0. 0. 1. 2. 0. 0.]
 [0. 0. 1. 2. 1. 0.]]
[[0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0.]
 [0. 0. 2. 2. 0. 0.]
 [0. 0. 2. 2. 0. 0.]
 [0. 0. 1. 1. 0. 0.]
 [0. 0. 1. 2. 0. 0.]
 [0. 0. 1. 2. 1. 0.]]
[[0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0.]
 [0. 0. 2. 2. 0. 0.]
 [0. 0. 2. 2. 0. 0.]
 [0. 0. 1. 1. 0. 0.]
 [0. 0. 1. 2. 1. 0.]
 [0. 0. 1. 2. 1. 0.]]
[[0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 2. 0. 0.]
 [0. 0. 2. 2. 0. 0.]
 [0. 0. 2. 2. 0. 0.]
 [0. 0. 1. 1. 0. 0.]
 [0. 0. 1. 2. 1. 0.]
 [0. 0. 1. 2. 1. 0.]]
[[0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 2. 0. 0.]
 [0. 0. 2. 2. 0. 0.]
 [0. 0. 2. 2. 0. 0.]
 [0. 0. 1. 1. 0. 0.]
 [0. 0. 1. 2. 1. 0.]
 [0. 0. 1. 2. 1. 0.]]
[[0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 2. 0. 0.]
 [0. 0. 2. 2. 0. 0.]
 [0. 0. 2. 2. 0. 0.]
 [0. 0. 1. 1. 2. 0.]
 [0. 0. 1. 2. 1. 0.]
 [0. 0. 1. 2. 1. 0.]]
[[0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 2. 0. 0.]
 [0. 0. 2. 2. 0. 0.]
 [0. 0. 2. 2. 1. 0.]
 [0. 0. 1. 1. 2. 0.]
 [0. 0. 1. 2. 1. 0.]
 [0. 0. 1. 2. 1. 0.]]
[[0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 2. 0. 0.]
 [0. 0. 2. 2. 0. 0.]
 [0. 0. 2. 2. 1. 0.]
 [0. 0. 1. 1. 2. 0.]
 [0. 0. 1. 2. 1. 0.]
 [0. 2. 1. 2. 1. 0.]]
[[0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 2. 0. 0.]
 [0. 0. 2. 2. 0. 0.]
 [0. 0. 2. 2. 1. 0.]
 [0. 0. 1. 1. 2. 0.]
 [0. 0. 1. 2. 1. 0.]
 [0. 2. 1. 2. 1. 1.]]
[[0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 2. 0. 0.]
 [0. 0. 2. 2. 0. 0.]
 [0. 0. 2. 2. 1. 0.]
 [0. 0. 1. 1. 2. 0.]
 [0. 0. 1. 2. 1. 2.]
 [0. 2. 1. 2. 1. 1.]]
```

Challenge task [+ 1 bonus point]

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 [1 point]

Undergraduate students: This is a bonus task you can attempt if you like [+1 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 [15]:
```
# Your code/ answer goes here.
```

Best First Move

How would you determine what the best first move is? You can use Pure Monte Carlo Search or any algorithms that you have implemented above.

In [16]:
```
# Your code/ answer goes here.
#The best move is generally start from the middle of the board
```