CSCI 3202, Spring 2023

Final Project

Project Due: Wednesday, May 3 at 9:00 PM

Proposals Due: Wednesday, April 12 at 9:00 PM

You have two options for completing your final project for this course.

Option 1

The first option is presented in this notebook and involves implementing a Connect Four game with AB pruning and A* as player strategies.

Option 2

The second option is to design your own project that includes any of the algorithms we've discussed throughout the semester, or an algorithm that you're interested in learning that we haven't discussed in class. Your project also needs to include some kind of analysis of how it performed on a specific problem. If you're interested in the design your own project option, you need to discuss your idea with one of the course instructors to get approval. If you do a project without getting approval, you will receive a 0 regardless of the quality of the project.

The rules:

1. Choose EITHER the given problem to submit OR choose your own project topic.

2. If you choose your own project topic, please adhere to the following guidelines:

- Send an email to the course instructor before Wednesday, April 12, with a paragraph description of your project. I will respond in 24 hours.
- The project can include an algorithm we've discussed throughout the semester or an algorithm that you're been curious to learn. Please don't recycle a project that you did in another class.
- If you do your own project without prior approval, you will receive a 0 for this project.
- Your project code, explanation, and results must all be contained in a Jupyter notebook.

1. All work, code and analysis must be **your own**.
2. You may use your course notes, posted lecture slides, textbook, in-class notebooks and homework solutions as resources. You may also search online for answers to general knowledge questions, like the form of a probability distribution function, or how to perform a particular

operation in Python. You may not use entire segments of code as solutions to any part of this project, e.g. if you find a Python implementation of policy iteration online, you can't use it.
3. You may **not** post to message boards or other online resources asking for help.
4. **You may not collaborate with classmates or anyone else.**
5. This is meant to be like a coding portion of an exam. So, we will be much less helpful than we typically are with homework. For example, we will not check answers, help debug your code, and so on.
6. If you have a question, post it first as a **private** Piazza message. If we decide that it is appropriate for the entire class, then we will make it a public post (and anonymous).
7. If any part of the given project or your personal project is left open-ended, it is because we intend for you to code it up however you want. Our primary concern is with the plots/analysis that your code produces. Feel free to ask clarifying questions though.

Violation of these rules will result in an **F** and a trip to the Honor Code council.

By writing your name below, you agree to abide by these rules:

Your name: JULIA TRONI

Some useful packages and libraries:

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import colors
from collections import deque
import heapq
import unittest
from scipy import stats
import copy as cp
from time import time
```

Problem 1: Game Theory - Playing "intelligent" Connect Four (100 points total)

Connect Four is a two-player game where the objective is to get four pieces in a row - horizontally, vertically, or diagonally. Check out this video if you're unfamiliar with the game: https://www.youtube.com/watch?v=utXzIFEVPjA.

(1a) Defining the Connect Four class structure (10 points)

We've provided the humble beginnings of a Connect Four game. You need to fill in this class structure for Connect Four using what we did during class as a guide, and then implement min-max

search with AB pruning, and A* search with at least one heuristic function. The class structure includes the following:

- `moves` is a list of columns to represent which moves are available. Recall that we are using matrix notation for this, where the upper-left corner of the board, for example, is represented at (1,1).
- `result(self, move, state)` returns a ***hypothetical*** resulting `State` object if `move` is made when the game is in the current `state`. Note that when a 'move' is made, the column must have an open slot and the piece must drop to the lowest row.
- `compute_utility(self, move, state)` calculates the utility of `state` that would result if `move` is made when the game is in the current `state`. This is where you want to check to see if anyone has gotten `nwin` in a row
- `game_over(self, state)` returns `True` if the game in the given `state` has reached a terminal state, and `False` otherwise.
- `utility(self, state, player)` returns the utility of the current state if the player is Red and $-1\cdot$ utility if the player is Black.
- `display(self)` is a method to display the current game `state`. You get it for free because this would be super frustrating without it.
- `play_game(self, player1, player2)` returns an integer that is the utility of the outcome of the game (+1 if Red wins, 0 if draw, -1 if Black wins). `player1` and `player2` are functional arguments that we will deal with in parts **1b** and **1d**.

Some notes:

- Assume Red always goes first.
- Do **not** hard-code for 6x7 boards.
- You may add attributes and methods to these classes as needed for this problem.

In [2]:

```
class State:
    def __init__(self, moves):
        self.to_move = 'R'
        self.utility = 0
        self.board = {}
        self.moves = moves

class ConnectFour:
    def __init__(self, nrow=6, ncol=7, nwin=4):
        self.nrow = nrow
        self.ncol = ncol
        self.nwin = nwin

        """ **Important**:you can play only the lowest empty row in the column."""
        moves = [(row,col) for row in range(1, nrow + 1) for col in range(1, ncol + 1)]
        self.state = State(moves)

    def legal_moves(self, state):
        legmoves = []
        for col in range(1, self.ncol + 1):
                for row in range(1, self.nrow + 1):
                    if (row, col) in state.moves:
                        # check if this is the lowest empty row in the column
```

```
                    if row == self.nrow or (row + 1, col) not in state.moves:
                        legmoves.append((row, col))
                        break  # move to next column
        return legmoves


    def result(self, move, state):
        '''
        What is the hypothetical result of move `move` in state `state` ?
        move  = (row, col) tuple where player will put their mark (R or B)
        state = a `State` object, to represent whose turn it is and form
                the basis for generating a **hypothetical** updated state
                that will result from making the given `move`

        Note that when a 'move' is made, the column must have an open slot
        and the piece must drop to the lowest row.

       '''

        '''Return a list of legal moves in the given state.
        A legal move i s any open column (lowest row in that column).'''


        # your code goes here...

        # piece must drop to the lowest row
        for row in range(self.nrow,0,-1):
            if (row,move[1]) in state.moves:
                move = (row,move[1])
                break

         # Don't do anything if the move isn't a legal one
        if (move not in state.moves):
            return state

        # Return a copy of the updated state:
        #   compute utility, update the board, remove the move, update whose turn
        new_state = cp.deepcopy(state)
        new_state.utility = self.compute_utility(move, state)
        new_state.board[move] = state.to_move
        new_state.moves.remove(move)
        new_state.to_move = ('R' if state.to_move == 'B' else 'B')
        return new_state

    def compute_utility(self, move, state):
        '''
        What is the utility of making move `move` in state `state`?
        If 'R' wins with this move, return 1;
        if 'B' wins return -1;
        else return 0.
        '''

        # your code goes here...
        row, col = move
        player = state.to_move

        # create a hypothetical copy of the board, with 'move' executed
        board = cp.deepcopy(state.board)
        board[move] = player
```

```
        # what are all the ways 'player' could with with 'move'?

        # check for row-wise win
        in_a_row = 0
        best_in_a_row=0
        for c in range(1,self.ncol+1):
            if board.get((row,c))==player:
                in_a_row += 1
            if in_a_row > best_in_a_row:
                best_in_a_row=in_a_row
            else: in_a_row = 0

        # check for column-wise win
        in_a_col = 0
        best_in_a_col=0
        for r in range(1,self.nrow+1):
            if board.get((r,col))==player:
                in_a_col +=1
            if in_a_col> best_in_a_col:
                best_in_a_col=in_a_col
            else: in_a_col = 0

        # check for NW->SE diagonal win
        in_a_diag1 = 0
        best_in_a_diag1=0
        for r in range(row,0,-1):
            if board.get((r,col-(row-r)))==player: in_a_diag1+=1
            if in_a_diag1>best_in_a_diag1: best_in_a_diag1=in_a_diag1
        for r in range(row+1,self.nrow+1):
            if board.get((r,col-(row-r)))==player:in_a_diag1+=1
            if in_a_diag1>best_in_a_diag1: best_in_a_diag1=in_a_diag1
            else: in_a_diag1=0

        # check for SW->NE diagonal win
        in_a_diag2 = 0
        best_in_a_diag2=0
        for r in range(row,0,-1):
            if board.get((r,col+(row-r)))==player: in_a_diag2 +=1
            if in_a_diag2> best_in_a_diag2: best_in_a_diag2=in_a_diag2
            else: in_a_diag2 =0
        for r in range(row+1,self.nrow+1):
            if board.get((r,col+(row-r)))==player: in_a_diag2 += 1
            if in_a_diag2> best_in_a_diag2: best_in_a_diag2=in_a_diag2
            else: in_a_diag2 =0

        if self.nwin in [best_in_a_row, best_in_a_col, best_in_a_diag1, best_in_a_diag2
            return 1 if player=='R' else -1
        else:
            return 0

    def game_over(self, state):
        '''game is over if someone has won (utility!=0) or there
        are no more moves left'''

        # your code goes here...
        return state.utility!=0 or len(state.moves)==0


    def utility(self, state, player):
        '''Return the value to player; 1 for win, -1 for loss, 0 otherwise.'''
```

```
        # your code goes here...
        return state.utility if player=='R' else -state.utility


    def display(self):
        board = self.state.board
        for row in range(1, self.nrow + 1):
            for col in range(1, self.ncol + 1):
                print(board.get((row, col), '.'), end=' ')
            print()


    def play_game(self, player1, player2):
        '''Play a game of Connect Four!'''

        # your code goes here...
        turn_limit = self.nrow*self.ncol  # limit in case of buggy code
        turn = 0
        while turn<=turn_limit:
            for player in [player1, player2]:
                turn += 1
                move = player(self)
                self.state = self.result(move, self.state)
                if self.game_over(self.state):
                    #self.display()
                    return self.state.utility
```

(1b) Define a random player (10 points)

Define a function `random_player` that takes a single argument of the `ConnectFour` class and returns a random move out of the available legal moves in the `state` of the `ConnectFour` game.

In your code for the `play_game` method above, make sure that `random_player` could be a viable input for the `player1` and/or `player2` arguments.

In [3]:
```
def random_player(game):
    '''A player that chooses a legal move at random out of all
    available legal moves in ConnectFour state argument'''

    # your code goes here...
    """ **Important**:you can play only the lowest empty row in the column."""
    possible_actions=game.state.moves
    return possible_actions[np.random.randint(low=0, high=len(possible_actions))]
```

We know from experience and/or because I'm telling you right now that if two `random_player`s play many games of ConnectFour against one another, whoever goes first will win about 55% of the time. Verify that this is the case by playing at least 1,000 games between two random players. Report the proportion of the games that the first player has won.**(Chris: is this true for TicTacToe, or Connect Four)**

"Unit tests": If you are wondering how close is close enough to 55%, I simulated 100 tournaments of 1,000 games each. The min-max range of win percentage by the first player was 52-59%.

In [4]:
```
# 1000 games between two random players

# Your code here

niter = 1000
wins = 0
draws = 0
losses = 0
for k in range(niter):
    cf = ConnectFour(3,4,3)
    out = cf.play_game(random_player, random_player)
    if out==1:
        wins += 1
    elif out==-1:
        losses += 1
    else:
        draws += 1

print('First-player winning percentage = {}'.format(wins/niter))
print('First-player losing percentage = {}'.format(losses/niter))
print('First-player draw percentage = {}'.format(draws/niter))
```

```
First-player winning percentage = 0.593
First-player losing percentage = 0.35
First-player draw percentage = 0.057
```

(1c) What about playing randomly on different-sized boards? (20 points)

What does the long-term win percentage appear to be for the first player in a 10x10 ConnectFour tournament, where 4 marks must be connected for a win? Support your answer using a simulation and printed output, similar to **1b**.

Also: The win percentage should have changed substantially. Did the decrease in wins turn into more losses for the first player or more draws? Write a few sentences explaining the behavior you observed. *Hint: think about how the size of the state space has changed.*

In [5]:
```
# 1000 games between two random players

# Your code here
niter = 1000
wins = 0
draws = 0
losses = 0
for k in range(niter):
    cfbig= ConnectFour(nrow=10, ncol=10, nwin=4)
    out = cfbig.play_game(random_player, random_player)
    #print(out)
    #print("")
    if out==1:
        wins += 1
    elif out==-1:
        losses += 1
    else:
        draws += 1
```

```
print('First-player winning percentage = {}'.format(wins/niter))
print('First-player losing percentage = {}'.format(losses/niter))
print('First-player draw percentage = {}'.format(draws/niter))
```

```
First-player winning percentage = 0.516
First-player losing percentage = 0.484
First-player draw percentage = 0.0
```

On a 3x4 board of connect 3, the long term win percentage for first player is 55-60%. But for a 10x10 board of connect 4 it decreases to 52-55% wins for first player. The decrease in wins turn into many more losses and no ties (decrease in tie percentage).

This is possible because the chance of actually receiving a tie being much lower due to the larger size of the boardsince it is more difficult to tie with such a large board and small win condition.

(1d) Define an alpha-beta player (20 points)

Alright. Let's finally get serious about our Connect Four game. No more fooling around!

Craft a function called `alphabeta_player` that takes a single argument of a `ConnectFour` class object and returns the minimax move in the `state` of the `ConnectFour` game. As the name implies, this player should be implementing alpha-beta pruning as described in the textbook and lecture.

Note that your alpha-beta search for the minimax move should include function definitions for `max_value` and `min_value` (see the aggressively realistic pseudocode from the lecture slides).

In your code for the `play_game` method above, make sure that `alphabeta_player` could be a viable input for the `player1` and/or `player2` arguments.

In [6]:
```
# Search game to determine best action; use alpha-beta pruning.
def alphabeta_player(game):
    def max_value(state, alpha, beta):
        player=game.state.to_move
        if game.game_over(state):
            return game.utility(state,player)
        v= -np.inf

        for a in state.moves:
            v= max(v, min_value(game.result(a, state), alpha, beta))
            if v >= beta:
                return v #pruning
            alpha = max(alpha, v)
        return v

    def min_value(state, alpha, beta):
        player=game.state.to_move
        if game.game_over(state):
            return game.utility(state,player)
        v= +np.inf
        for a in state.moves:
            v= min(v, max_value(game.result(a, state), alpha, beta))
            if v <= alpha:
                return v #pruning
```

```
            beta=min(beta,v)
        return v

    best_score = -np.inf
    beta=np.inf
    best_action = None

    for a in game.state.moves:
        v = min_value(game.result(a, game.state), best_score, beta)

        if v > best_score:
            best_score = v
            best_action = a
    return best_action
```

In [7]:

```
# Another implementation that does not kill the kernel
#This is technically depth limited
#it does not kill the kernel but it is incorrect for alphabeta vs alphabeta player


def depthlimited_alphabeta_player(game, d=6, cutoff_test=None, eval_fn=None):
    """As in [Figure 5.7], this version searches all the way to the leaves."""

    # Get the player whose turn it is
    player = game.state.to_move


    def max_value(state, alpha, beta, depth):
        if cutoff_test(state, depth):
            return eval_fn(state)
        v= -np.inf

        legal = []
        legal = game.legal_moves(game.state)

        for a in legal:
            v= max(v, min_value(game.result(a, state), alpha, beta, depth+1))
            if v >= beta:
                return v #pruning
            alpha = max(alpha, v)
        return v

    def min_value(state, alpha, beta, depth):
        if cutoff_test(state, depth):
            return eval_fn(state)

        v= +np.inf

        legal = []
        legal = game.legal_moves(game.state)

        for a in legal:
            v = min(v, max_value(game.result(a, state), alpha, beta, depth+1))
            if v <= alpha:
                return v #pruning
            beta = min(beta, v)

        return v
```

```
    # Body of alpha_beta_search:
    cutoff_test = (cutoff_test or (lambda state, depth: depth > d or game.game_over(sta
    eval_fn = eval_fn or (lambda state: game.utility(state, player))
    best_score = -np.inf
    beta = np.inf
    best_action = None

    legal = []
    legal = game.legal_moves(game.state)

    count=0
    for a in legal:
        count+=1
        v = min_value(game.result(a, game.state), best_score, beta, 1)
        if v > best_score:
            best_score = v
            best_action = a
    return best_action
```

Verify that your alpha-beta player code is working appropriately through the following tests, using a standard 6x7 ConnectFour board. Run **10 games for each test**, and track the number of wins, draws and losses. Report these results for each case.

1. An alpha-beta player who plays first should never lose to a random player who plays second.
2. Two alpha-beta players should always draw. One player is the max player and the other player is the min player.---> REMOVED per Jim's announcement

Nota bene: Test your code with fewer games between the players to start, because the alpha-beta player will require substantially more compute time than the random player. This is why I only ask for 10 games, which still might take a minute or two. You are welcome to run more than 10 tests if you'd like.

In [8]:
```
# 1. An alpha-beta player who plays first should never lose to a random player who play
niter = 10
wins = 0
draws = 0
losses = 0
for k in range(niter):
    cf = ConnectFour(3,4,3)
    out = cf.play_game(alphabeta_player, random_player)
    if out==1:
        wins += 1
    elif out==-1:
        losses += 1
    else:
        draws += 1

print('First-player winning percentage = {}'.format(wins/niter))
print('First-player losing percentage = {}'.format(losses/niter))
print('First-player draw percentage = {}'.format(draws/niter))
```

```
First-player winning percentage = 1.0
First-player losing percentage = 0.0
First-player draw percentage = 0.0
```

(1e) What has pruning ever done for us? (10 points)

Calculate the number of number of states expanded by the minimax algorithm, **with and without pruning**, to determine the minimax decision from the initial 6x7 ConnectFour board state. This can be done in many ways, but writing out all the states by hand is **not** one of them (as you will find out!).

Then compute the percent savings that you get by using alpha-beta pruning. i.e. Compute $\frac{\text{Number of nodes expanded with pruning}}{\text{Number of nodes expanded with minimax}}$

Write a sentence or two, commenting on the difference in number of nodes expanded by each search.

In [9]:

```
pruningExpanded=0
minmaxExpanded=0

def alpha_beta_pruning(game):
    def max_value(state, alpha, beta):
        player=game.state.to_move
        if game.game_over(state):
            return game.utility(state,player)
        v= -100

        for a in state.moves:
            global pruningExpanded
            pruningExpanded+=1
            v= max(v, min_value(game.result(a, state), alpha, beta))
            if v >= beta:
                return v #pruning occurs here
            alpha = max(alpha, v)
        return v

    def min_value(state, alpha, beta):
        player=game.state.to_move
        if game.game_over(state):
            return game.utility(state,player)
        v= +100

        for a in state.moves:
            global pruningExpanded
            pruningExpanded+=1
            v= min(v, max_value(game.result(a, state), alpha, beta))
            if v <= alpha:
                return v #pruning occurs here
            beta=min(beta,v)
        return v

    best_score = -100
    beta=100
    best_action = None

    for a in game.state.moves:
        global pruningExpanded
        pruningExpanded+=1

        v = min_value(game.result(a, game.state), best_score, beta)
        if v > best_score:
```

```
            best_score = v
            best_action = a
    return best_action
```

In [10]:

```
pruningExpanded=0
minmaxExpanded=0

def min_max(game):
    def max_value(state, alpha, beta):
        player=game.state.to_move
        if game.game_over(state):
            return game.utility(state,player)
        v= -100

        for a in state.moves:
            global minmaxExpanded
            minmaxExpanded+=1
            v= max(v, min_value(game.result(a, state), alpha, beta))
            alpha = max(alpha, v)
        return v

    def min_value(state, alpha, beta):
        player=game.state.to_move
        if game.game_over(state):
            return game.utility(state,player)
        v= +100

        for a in state.moves:
            global minmaxExpanded
            minmaxExpanded+=1
            v= min(v, max_value(game.result(a, state), alpha, beta))
            beta=min(beta,v)
        return v

    best_score = -100
    beta=+100
    best_action = None

    for a in game.state.moves:
        global minmaxExpanded
        minmaxExpanded+=1

        v = min_value(game.result(a, game.state), best_score, beta)
        if v > best_score:
            best_score = v
            best_action = a
    return best_action
```

In [11]:

```
cf = ConnectFour(3,4,3)
alpha_beta_pruning(cf)

print('alpha_beta_pruning = {}'.format(pruningExpanded))
```

```
alpha_beta_pruning = 32592
```

In [ ]:

```
cf = ConnectFour(3,4,3)
min_max(cf) #NOTE!!! This took 30+ minutes to run on my shitty laptop

print('min_max = {}'.format(minmaxExpanded))



print("Percent savings from pruning: ", (((pruningExpanded)/minmaxExpanded)* 100), "%")
```

With pruning in alphabeta we will expand SIGNIFICANTLY LESS nodes than without pruning with minimax. The number of states that did not get pruned is MUCH larger than the number expanded with pruning, which shows that alpha beta is far more efficient that minimax

(2) A* Search

In Part II of this project, you need to implement a player strategy to employ A *Search in order to win at ConnectFour. To test your A* player, play 10 games against the random player and 10 games against the AB pruning player.

Write an explanation of this strategy's implementation and performance in comparison to the random player and the AB Pruning player from **1d**.

A lot of the code that you wrote for the minimax player and the Connect Four game structure can be reused for the A *player. However, you will need to write a new utility function for A* that considers the path cost and heuristic cost.

(2a) Define a heuristic function (20 points)

Your A* player will need to use a heuristic function. You have two options for heurstics: research an existing heuristic for Connect Four, or games similar to Connect Four, and use that. Or, design your own heuristic. Write a one-paragraph description of the heuristic you're using, including a citation if you used an existing heuristic.

In [12]:

```
from heapq import heappush, heappop


#heavily relied on AIMA search repo and class notebooks
def astar_player(game):

    def heuristic_utility(move, state):
        player = state.to_move

        board = cp.deepcopy(state.board)
        board[move] = player
        row=game.nrow
        col=game.ncol
        counter=0

        # check for row-wise win
        for c in range(1,game.ncol+1):
            if board.get((row,c))==player:counter +=1
            else: break
```

```
            if counter >= game.nwin:
                return 0


        # check for column-wise win
        for r in range(1,game.nrow+1):
            if board.get((r,col))==player:counter +=1
            else: break
            if counter >= game.nwin:
                return 0

        # check for NW->SE diagonal win
        in_a_diag1 = 0
        best_in_a_diag1=0
        for r in range(row,0,-1):
            if board.get((r,col-(row-r)))==player: counter +=1
            else: break
            if counter >= game.nwin:
                return 0
        for r in range(row+1,game.nrow+1):
            if board.get((r,col-(row-r)))==player:counter +=1
            else: break
            if counter >= game.nwin:
                return 0
        # check for SW->NE diagonal win
        in_a_diag2 = 0
        best_in_a_diag2=0
        for r in range(row,0,-1):
            if board.get((r,col+(row-r)))==player: counter +=1
            else: break
            if counter >= game.nwin:
                return 0
        for r in range(row+1,game.nrow+1):
            if board.get((r,col+(row-r)))==player: counter +=1
            else: break
            if counter >= game.nwin:
                return 0

        return counter


    #number of states on board left
    def spots(move, state):
        return len(state.board)

    def full_heuristic(move, state):
        total = spots(move, state) + heuristic_utility(move, state)
        return total

    #A star algorithm
    def aStarSearch(state):
        frontier = []
        heappush(frontier, (full_heuristic(None, state), None, state))
        explored = set()
        while frontier:
            _, move, state = heappop(frontier)
            if state in explored:
                continue
            explored.add(state)
            if state.to_move != game.state.to_move:
```

```
                for move in state.moves:
                    if game.game_over(game.result(move, state)):
                        return move
                    cost = full_heuristic(move, state)
                    state = game.result(move, state)
                    heappush(frontier, (cost, move, state))
            return (move, state)

    return aStarSearch(game.state)
```

I implemented a heuristic that estimates the number of moves required to form a winning sequence (see function 'heuristic_utility'). This heuristic is used to calculate the total cost of a move by adding this heuristic value to the size of the current board state.

I use this heuristic in a A* search using a priority queue implemented with a heap, where nodes are explored in increasing order of their total cost.

(2b) Compare A* to other algorithms (10 points)

Next, play 10 games of Connect Four using your A* player and a random player and 10 games against the AB pruning player. In four or five paragraphs, report on the outcome. Did one player win more than the other? How often was the game a draw? How many moves did each player make? Were there situations where one player appeared to do better than the other? Given the outcome, are there other heuristics you would like to implement?

In [13]:
```
#1000 games of Connect Four using your A* player and a random player
#** Note: I tested for 1000 iterations to get more consistency

niter = 1000
wins = 0
draws = 0
losses = 0
for k in range(niter):
    cf = ConnectFour(3,4,3)
    out = cf.play_game(astar_player, random_player)
    if out==1:
        wins += 1
    elif out==-1:
        losses += 1
    else:
        draws += 1

print('A Star-player winning percentage = {}'.format(wins/niter))
print('A star-player losing percentage (aka Random Player wins) = {}'.format(losses/nit
print('Draw percentage = {}'.format(draws/niter))
```

```
A Star-player winning percentage = 0.25
A star-player losing percentage (aka Random Player wins) = 0.039
Draw percentage = 0.711
```

In [14]:
```
#10 games of Connect Four using your A* player and a alphabeta player
niter = 10
wins = 0
draws = 0
losses = 0
```

```
for k in range(niter):
    cf = ConnectFour(3,4,3)
    out = cf.play_game(astar_player, alphabeta_player)
    if out==1:
        wins += 1
    elif out==-1:
        losses += 1
    else:
        draws += 1

print('A Star-player winning percentage = {}'.format(wins/niter))
print('A star-player losing percentage (aka Alpha Beta Player wins) = {}'.format(losses
print('Draw percentage = {}'.format(draws/niter))
```

```
A Star-player winning percentage = 1.0
A star-player losing percentage (aka Alpha Beta Player wins) = 0.0
Draw percentage = 0.0
```

For both Astar vs Random and Astar vs Alpha Beta, Astar player won more than the other.

Especially Astar vs Alpha Beta, the Astar player won 100% of the time.

For Astar vs Random, the Random player never (or very rarely) won, but the two player generally tied more than 55% of the time, which I found interesting.

We discussed other heuristics like Manhattan distance in class, and I think it would be interesting to implement that. https://iopscience.iop.org/article/10.1088/1742-6596/1898/1/012047/pdf

There are also numberous papers with interesting ideas that would be neat to dig into like this one that considers a direction based approach https://pdf.sciencedirectassets.com/280203/1-s2.0-S1877050915X00068/1-s2.0-S1877050915004743/main.pdf?X-Amz-Security-Token=IQoJb3JpZ2luX2VjEMn%2F%2F%2F%2F%2F%2F%2F%2F%2F%2FwEaCXVzLWVhc3QtMSJGMEQCI
Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Date=20230503T091022Z&X-Amz-SignedHeaders=host&X-Amz-Expires=300&X-Amz-Credential=ASIAQ3PHCVTY7PZJI32I%2F20230503%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Signature=47abb5a2b70d3989fd166ea672e70e11e95159e2b8318f655e4bac62a39eff41&hash=a24661524ebb82f-4cab-45b3-9cba-1f098c2c9d3a&sid=1b502b8a920ba64a7e59eac44fc779071c6dgxrqa&type=client&tsoh=d3d3LnNjaWV