CS 7320 HW3 Submission Template

Name: Varun Singh
ID: 4900 6872

Part 1:

Test1 Output:

```
PASS Test 1

PASS Test 2

PASS Test 3  4x4 array
```

Test2 Output:

```
PASS Test 1 No Win

PASS Test 2  column win

PASS Test 3  row win

PASS Test 4  diag win

PASS Test 5  diag2 win

PASS Test 6  diag2 win 4x4
```

Test3 Output:

```
PASS Test 1 Non Terminal Board

PASS Test 2  Terminal Board

PASS Test 3  Terminal Board

PASS Test 4  Terminal Board

PASS Test 5  Terminal Board

PASS Test 6  Terminal Board
```

Part 2.

Run MiniMax Results

Board	Score	# Boards	Time
b2	-1	941	0.016200 seconds
b3	0	422,074	6.419683 seconds
b4	0	4,861,625	143.272166 second

```
# MiniMax - Get score for board

import math
import numpy as np
import time
import random

from copy import copy
import time
COUNT = 0    # use the COUNT variable to track number of boards explored
max_depth = 0  # adjust this for each board
def showBoard(board):
    # displays rows of board
    strings = ["" for i in range(board.shape[0])]
    idx = 0
    for row in board:
        for cell in row:
            if cell == 1:
                s = 'X'
            elif cell == -1:
                s = 'O'
            else:
                s = '_'

            strings[idx] += s
        idx += 1

    # display final board
    for s in strings:
        print(s)

def get_board_one_line(board):
    # returns one line rep of a board
    import math
    npb_flat = board.ravel()
    stop = int(math.sqrt(len(npb_flat)))

    bstr = ''
    for idx in range(len(npb_flat)):
        bstr += (str(npb_flat[idx]) + ' ')
        if (idx + 1) % (stop) == 0:
            bstr += '|'
    return bstr

def evaluate(board):
    # replace with your code
    '''returns 1 for X win, -1 for O win, 0 for tie OR game in progress
        Using numpy functions to add values in rows and cols
        If we get a sum equal to size of row,col,diag (plus or minus)
         we have a winner
        '''
    # replace with your code
    for i in range(board.shape[0]):  # checking for winning condition in rows
        sum = 0
        for j in range(board.shape[1]):
            sum = sum + board[i][j]

        if sum == board.shape[0]:
            return 1
        elif sum == -board.shape[0]:
            return -1

    for i in range(board.shape[0]):  # checking for winning condition in columns
        sum = 0
        for j in range(board.shape[1]):
            sum = sum + board[j][i]

        if sum == board.shape[0]:
            return 1
        elif sum == -board.shape[0]:
            return -1
    sum = 0

    for i in range(board.shape[0]):  # checking win condition for forward
diagonal   \
        sum = sum + board[i][i]

    if sum == board.shape[0]:
        return 1
    elif sum == -board.shape[0]:
        return -1

    sum = 0
    for i in range(board.shape[0]):  # checking win condition for backward
diagonal  /
        for j in range(board.shape[0]):
            if (i + j) == board.shape[0] - 1:
                sum = sum + board[i][j]

    if sum == board.shape[0]:
        return 1
    elif sum == -board.shape[0]:
        return -1

    return 0

def is_terminal_node(board):
    # replace with your code
    global COUNT
    COUNT = COUNT+1

    if evaluate(board) != 0:
        return True
    flag = 0

    for i in range(board.shape[0]):
        for j in range(board.shape[1]):
            if board[i][j] == 0:
                flag = 1
                return False

    if flag == 0:  # flag value stays the same if the board is full
        return True


def get_child_boards(board, char):
    # replace with your code
    ''' numpy version '''
    if not char in ['X', 'O']:
        raise ValueError("get_child_boards: expecting char='X' or 'O' ")

    newval = -1
    if char == 'X': newval = 1

    child_list = []

    # add your code here
    for i in range(board.shape[0]):
        for j in range(board.shape[1]):
            duplicate_Board = board.copy()
            if duplicate_Board[i][j] == 0:
                duplicate_Board[i][j] = newval
                child_list.append(duplicate_Board)
                continue

    return child_list


def minimax(board, depth, maximizingPlayer):
    '''returns the value of the board
       0 (draw) 1 (win for X) -1 (win for O)
       Explores all child boards for this position and returns
       the best score given that all players play optimally
    '''

    if depth == 0 or is_terminal_node(board):
        return evaluate(board)

    if maximizingPlayer:  # max player plays X
        maxEva = -math.inf
        child_list = get_child_boards(board, 'X')
        for child_board in child_list:
            eva = minimax(child_board, depth-1, False)
            maxEva = max(maxEva, eva)
        return maxEva

    else:             # minimizing player
        minEva = math.inf
        child_list = get_child_boards(board, 'O')
        for child_board in child_list:
            eva = minimax(child_board, depth - 1, True)
            minEva = min(minEva, eva)
        return minEva

def run_minimax(board):
        # set max_depth to the number of blanks (zeros) in the board
    global max_depth
    max_depth = np.count_nonzero(board == 0)
    print(f"Running minimax w/ max depth {max_depth} for:")
    showBoard(board)
    number_of_X = 0
    number_of_O = 0

    for i in range(board.shape[0]):
            for j in range(board.shape[1]):
                if board[i][j] == 1:
                    number_of_X += 1
                elif board[i][j] == -1:
                    number_of_O +=1

    if number_of_X-number_of_O == 0:
        return True
    else:
        return False



def run_code_tests():
    '''
    b1 : expect win for X (1)  < 200 boards explored
    b1 = np.array([[1, 0, -1], [1, 0, 0], [-1, 0, 0]])

    In addtion to the board b1, run tests on the following
    boards:
       b2:  expect win for O (-1)  > 1000 boards explored
       b2 = np.array([[0, 0, 0], [1, -1, 1], [0, 0, 0]])

       b3: expect TIE (0)  > 500,000 boards explored; time around 20secs
       b3 = np.array([[0, 0, 0], [0, 0, 0], [0, 0, 0]])

       b4: expect TIE(0) > 7,000,000 boards;  time around 4-5 mins
       b4 = np.array(
        [[1, 0, 0, 0], [0, 1, 0, -1], [0, -1, 1, 0], [0, 0, 0, -1]])

    '''

    # Minimax for a board: evaluate the board
    #    expect win for X (1)  < 200 boards explored
    b1 = np.array([[1, 0, -1], [1, 0, 0], [-1, 0, 0]])
    b2 = np.array([[0, 0, 0], [1, -1, 1], [0, 0, 0]])
    b3 = np.array([[0, 0, 0], [0, 0, 0], [0, 0, 0]])
    b4 = np.array([[1, 0, 0, 0], [0, 1, 0, -1], [0, -1, 1, 0], [0, 0, 0, -1]])

    print(f"\n--------\nStart Board: \n{b1}")

    # set max_depth  to the number of blanks (zeros) in the board

    is_x_to_move = run_minimax(b4)
    print(is_x_to_move)

    start_time = time.time()
    # read time before and after call to minimax
    score = minimax(b4, max_depth, is_x_to_move)
    end_time = time.time()


    time_taken = end_time - start_time
    print("Time taken to run function: {:.6f} seconds".format(time_taken))
    print(f"count : {COUNT}")
    print (f"score : {score}")




if __name__ == '__main__':
    run_code_tests()
```

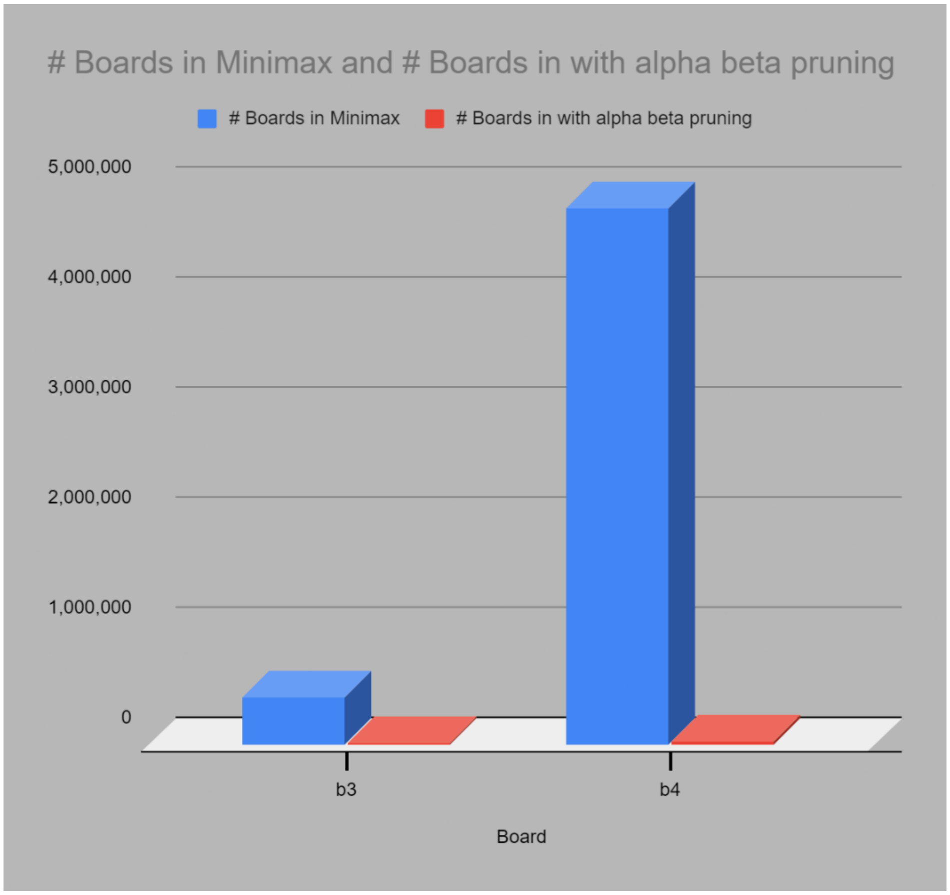
Part 3.

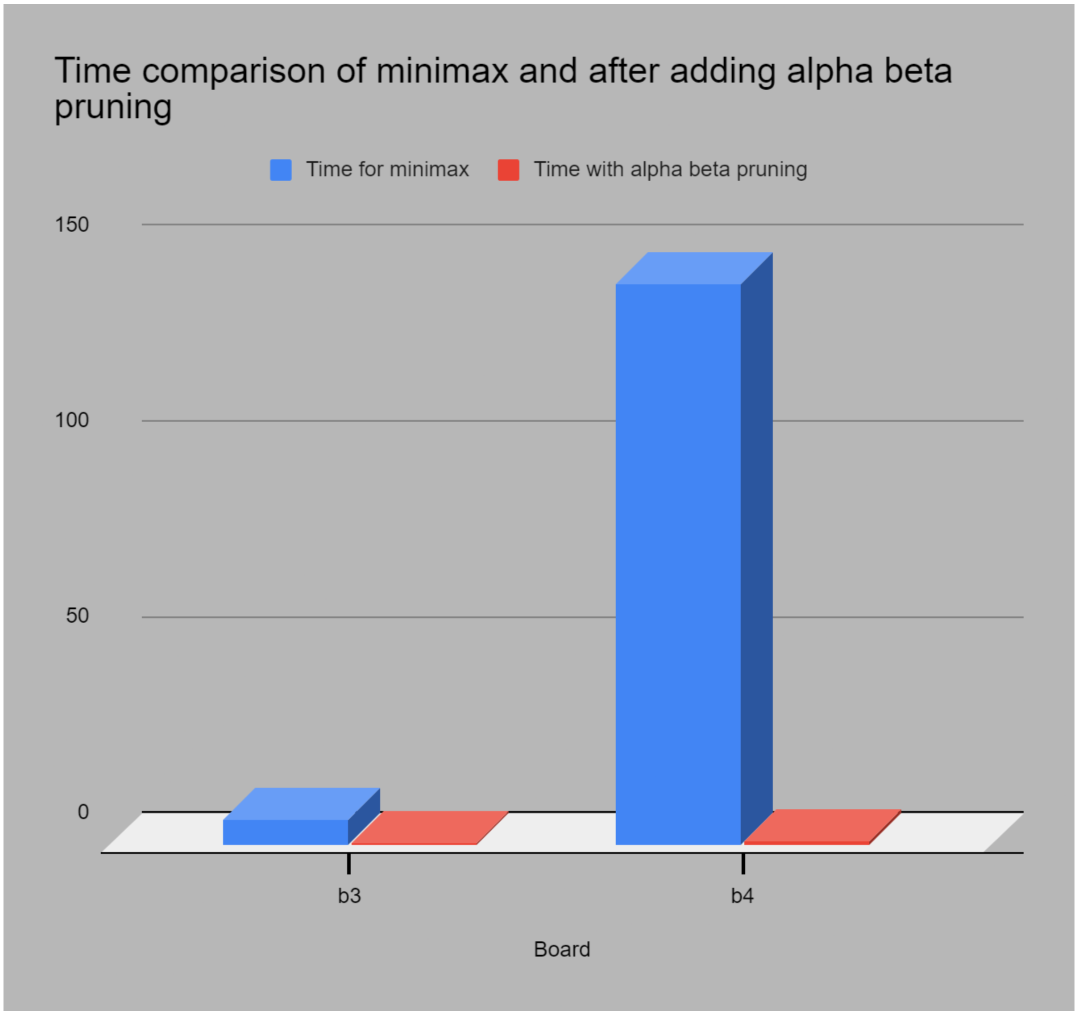
MiniMax Results

Board	Score	# Boards	Time
b3	0	422,074	6.419683 seconds
b4	0	4,861,625	143.272166 second

MiniMax w/ Alpha Beta Results

Board	Score	# Boards	Time
b3	0	15,021	0.238893 seconds
b4	0	21,747	0.604178 seconds




Time comparison of minimax and after adding alpha beta pruning
🟦 Time for minimax
🟥 Time with alpha beta pruning
150
100
50
0
b3
b4
Board

Extra Credit:

Describe your heuristic -

To improve the performance of the algorithm I came up with a couple of improvements where -

1. I tried to reduce the moves in rows, columns and diagonals that already have a -1 or O value in them. These rows or columns are basically useless for the player X since we can’t use these to win the game. So, we must use our turn in places that don’t waste our our move.

2. Another improvement I tried was to make sure if a column or row doesn’t have any -1 or O then how many 1’s does it have. So, if the current row or column already has some 1’s or X’s stored then we use that row or column to put our 1 or X.

Both these approaches are meant to reduce wastage of turns and maximize the impact each move has on the outcome of the game.

Show the code for your heuristic (not your entire program)

```
def get_child_boards(board, char):
    # replace with your code
    ''' numpy version '''
    if not char in ['X', 'O']:
        raise ValueError("get_child_boards: expecting char='X' or 'O' ")

    newval = -1
    if char == 'X': newval = 1

    child_list = []
    global rows_with_O      # Store rows which are unfavorable for player X
    global columns_with_O   # Store columns which are unfavorable for player X
    best_row_for_X = []     # Store rows which are favorable
    best_column_for_X = []  # Store columns which are favorable

    if newval == 1:
        for i in range(board.shape[0]):
            is_O_present_in_row = 0                         # flag variable
            for j in range(board.shape[1]):
                if board[i][j] == -1:               # if -1 or O is present that means this is not a favorable row
                    is_O_present_in_row = 1

            number_of_X = 0
            if is_O_present_in_row == 0:            # if -1 is absent then we count how many 1's are there in row
                for j in range(board.shape[1]):
                    if board[i][j] == 1:                    # now
                        number_of_X = number_of_X + 1
            else:
                number_of_X = -1                    # if -1 is present then we store -1 in the rows which are
                                             # favorable

            best_row_for_X.append(number_of_X)

        is_O_present_in_column = 0
        for i in range(board.shape[0]):
            for j in range(board.shape[1]):
                if board[j][i] == -1:               # if -1 is present then column is not favorable
                    is_O_present_in_column = 1

            number_of_X = 0
            if is_O_present_in_column == 0:         # if -1 is absent then column is favorable
                for j in range(board.shape[1]):
                    if board[j][i] == 1:            # now we count number of 1's
                        number_of_X = number_of_X + 1
            else:
                number_of_X = -1                    # if -1 is present then we store a negative value in the list

            best_column_for_X.append(number_of_X)

        for i in range(board.shape[0]):
            for j in range(board.shape[1]):
                duplicate_Board = board.copy()
                if best_row_for_X[j] > 0 and board[i][j] == 0:  #if j element of the best row list is greater than 0
                    duplicate_Board[i][j] = newval             # and the element on the board is 0
                    child_list.append(duplicate_Board)          # then we add a 1 and append the child_list
                    return child_list
                elif best_column_for_X[j] > 0 and board[i][j] == 0: # if j element of the best column list is greater
                    duplicate_Board[i][j] = newval                 # than 0 and the element in the board is 0 or
                    child_list.append(duplicate_Board)              # empty then we add a 1 to the board and append
                    return child_list
                elif board[i][j] == 0:                       # if we didnt find the best then we simply add 0 to
                    duplicate_Board[i][j] = newval             # the board and append the list
                    child_list.append(duplicate_Board)
                    return child_list
else:
    for i in range(board.shape[0]):                          # if the newval is -1 it means it is O's chance
        for j in range(board.shape[1]):                      # we simply add a -1 to the first empty space we get
            if board[i][j] == 0:                             # on the board
                duplicate_Board = board.copy()
                duplicate_Board[i][j] = newval
                child_list.append(duplicate_Board)
                return child_list


return child_list
```

In the **evaluate(board)** added this extra line of code -

```
    if len(rows_with_O) == board.shape[0] and len(columns_with_O) == board.shape[1] and is_Diagonal_Possible[0] == -1 and is_Diagonal_Possible[1] == -1:

        return 0
```

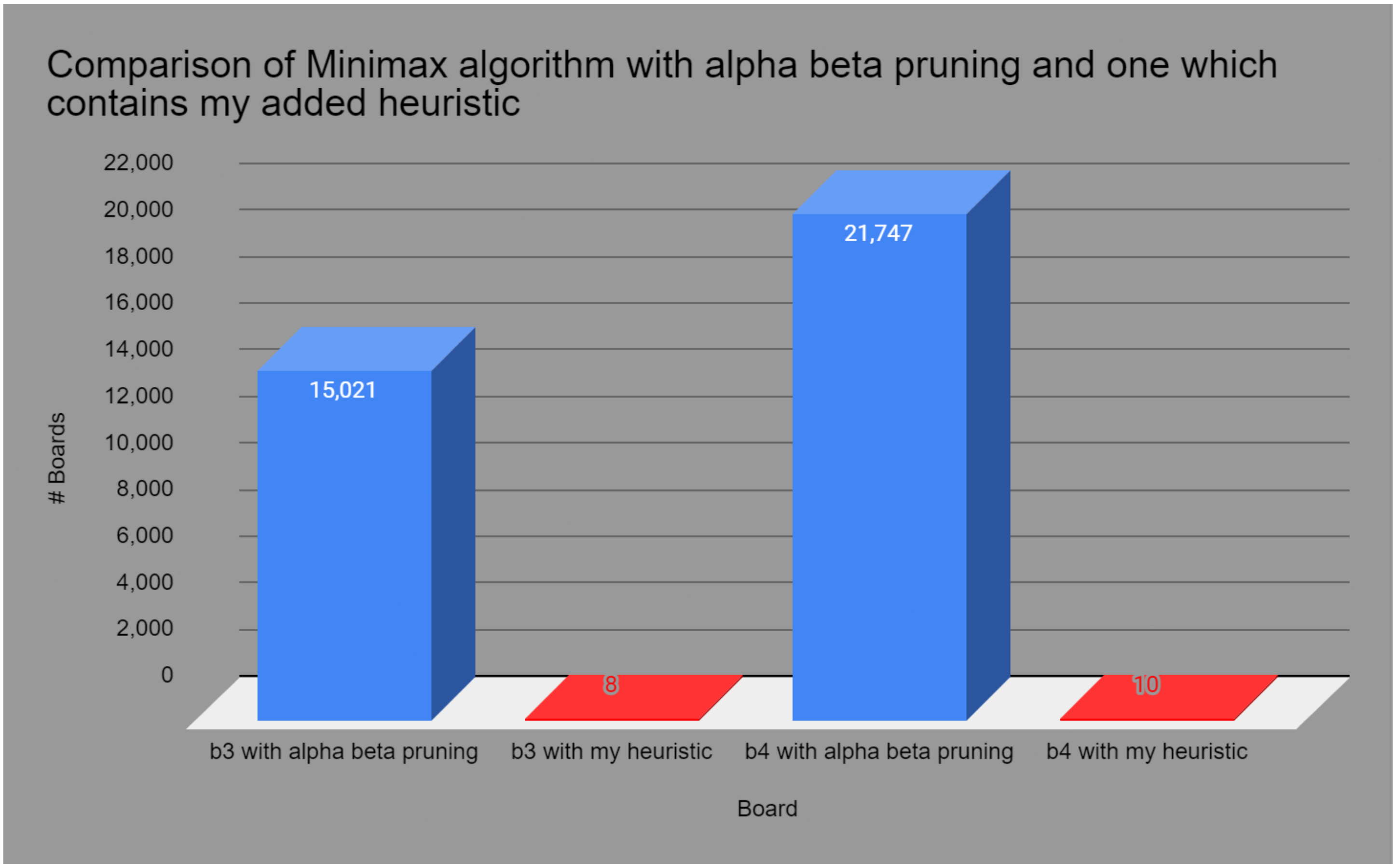
MiniMax w/ Alpha Beta

Board	Score	# Boards	Time
b3	0	15,021	0.238893 seconds
b4	0	21,747	0.604178 seconds

MiniMax w/ Alpha Beta Rw/ Heuristc

Board	Score	# Boards	Time
b3	1	8	0.000000 seconds
b4	0	10	0.001000 seconds

Bar Chart(s) Showing Comparison


Comparison of Minimax algorithm with alpha beta pruning and one which contains my added heuristic
# Boards
b3 with alpha beta pruning
b3 with my heuristic
b4 with alpha beta pruning
b4 with my heuristic
Board