Unbeatable Tic-Tac-Toe

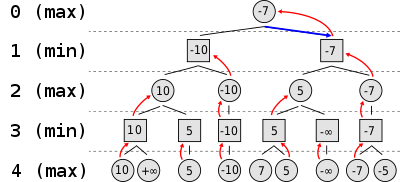
Mark Nakamae

Cal Poly SLO

August 2020

Description

This project looked to create an unbeatable opponent in the simple game of Tic-Tac-Toe. I accomplished this by implementing the [minimax algorithm](https://en.wikipedia.org/wiki/Minimax), in which moves are recursively scored based on the likelihood of a win for the computer. In other words, moves that led to a computer victory scored higher and moves that led to a human player win scored negatively. The algorithm recursively follows the possible moves sequences from both players, then returns the most optimal one. Since this project focused more on the implementation on minimax, I decided to make the game itself a very simple terminal text based implementation.



My Thoughts

I found [this page](https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/) from GeeksforGeeks to be a great reference for further understanding the minimax algorithm, and how to specifically apply it to the game I wanted, in this case Tic-Tac-Toe. My implementation did not seek for the most optimal win (least amount of turns), which could have been implemented if the code looked more at the depth of the recursion. My implementation does however ensure either a tie or a win for the computer. Given the scope of this project, this particular solution is possible since the game is able to recursively compute the scoring for each move with a reasonable amount of time. However in a larger scale game, the minimax solution may not be as applicable since there are too many different paths to account for, and could significantly slow down the program overall.

Code

**import** random

# print example board for user to select move

**def** print\_example\_board():

**print**("1 | 2 | 3")

**print**("---------")

**print**("4 | 5 | 6")

**print**("---------")

**print**("7 | 8 | 9")

**print**()

# check for a winner in the current board state

**def** check\_win(board):

# valid columns win, rows win, diagonals win respectively

valid = [[0, 3, 6], [1, 4, 7], [2, 5, 8],\

[0, 1, 2], [3, 4, 5], [6, 7, 8],\

[0, 4, 8], [2, 4, 6]]

# check for a winner in the current board state

**for** win **in** valid:

**if**(board[win[0]] == board[win[1]] **and** board[win[0]] == board[win[2]]\

**and** board[win[0]] != ' '):

**return** board[win[0]]

# tie found

**if**(' ' **not** **in** board):

**return** "tie"

# no winner found

**return** None

# add a valid move to the board

**def** add\_move(player, move, board):

# invalid move

**if**(move < 0 **or** move > 8):

**return** False

# valid move, board updated

**if**(board[move] == ' '):

board[move] = player

**return** True

#invalid move

**return** False

# print the current board state

**def** print\_board(board):

**print**(board[0] + " | " + board[1] + " | " + board[2])

**print**("---------")

**print**(board[3] + " | " + board[4] + " | " + board[5])

**print**("---------")

**print**(board[6] + " | " + board[7] + " | " + board[8])

**print**("\n=========\n")

# find best move for computer using minimax

**def** best\_move(board):

best\_score = -10

best\_spot = None;

**for** move **in** range(9):

**if**(board[move] == ' '):

board[move] = 'X'

score = minimax(board, 0, False)

board[move] = ' '

**if**(score > best\_score):

best\_score = score

best\_spot = move

**return** best\_spot

# recursively find the best move based on scorings

# a better score means the computer is more likely to win

**def** minimax(board, depth, isMax):

res = check\_win(board)

**if**(res != None):

**if**(res == "tie"):

**return** 0

**elif**(res == 'X'):

**return** 1

**elif**(res == 'O'):

**return** -1

# X's turn, find the best move

**if**(isMax == True):

best\_score = -10

**for** move **in** range(9):

**if**(board[move] == ' '):

board[move] = 'X'

score = minimax(board, depth + 1, False)

board[move] = ' '

best\_score = max(score, best\_score)

**return** best\_score

# O's turn, find the best move

**else**:

best\_score = 10

**for** move **in** range(9):

**if**(board[move] == ' '):

board[move] = 'O'

score = minimax(board, depth + 1, True)

board[move] = ' '

best\_score = min(score, best\_score)

**return** best\_score

**def** main():

# init

board = [' '] \* 9

move\_count = 0

# determine who moves first

**if**(random.randint(0, 1) == 0):

turn = 'X'

**else**:

turn = 'O'

# continue game while still available moves and no winner

**while**(move\_count < 9):

# print board state

print\_example\_board()

print\_board(board)

# ask player for a move and check if it is valid on their turn

**if**(turn == 'O'):

move = input("Select a move: ")

valid\_move = add\_move(turn, int(move) - 1, board)

# otherwise use minimax algorithm to find computer's best next move

**else**:

move = best\_move(board)

add\_move(turn, move, board)

valid\_move = True

# check for a winner

**if**(check\_win(board) == 'X' **or** check\_win(board) == '0'):

**print**("\n\n")

print\_board(board)

**print**(check\_win(board) + " wins!")

**return**

# increment turn counter and change player

**if**(valid\_move == True):

move\_count += 1

**if**(turn == 'X'):

turn = 'O'

**else**:

turn = 'X'

**print**("\n\n")

print\_board(board)

**print**("It's a tie!")

**if** \_\_name\_\_ == '\_\_main\_\_':

main()