**106408 – Artificial Intelligence and Expert Systems**

**Project Report**

**TIC-TAC-TOE**

**Project Members:**

Muhammad Yousuf ID: 10484

Taloot Ansari ID: 64190

Submitted to**:**​ Sir Siraj Munir

**Introduction:**

Shortly after, problems of this type grew into a challenge of great significance for development of one of today's hottest fields in computing - AI . a number of the best accomplishments in AI are achieved on the topic of strategic games - world champions in various strategic games have already been beaten by computers, e.g. in Chess, Checkers, Backgammon, and last (2016) even Go.

Although these programs are very successful, their way of creating decisions may be a lot different than that of humans. The bulk of those programs are supported efficient searching algorithms, and since recently on machine learning also .

**Defining Terms:**

Rules of the many of thosegames are defined by legal positions (or legal states) and legal moves for eachlegal position. For eachlegal position it'spossible to effectively determine all the legal moves. A number ofthe legal positions are starting positions and a feware ending positions.

The best thanks to describe these terms is employing a tree graph whose nodes are legal positions and whose edges are legal moves. The graph is directed since it doesn't necessarily mean that we'll be ready to withdrawexactly where we came from within the previous move, e.g. in chess a pawn can only proceed . This graph is nameda game tree. Moving down the sporttree represents one amongthe players making a move, and therefore thegame state changing from one legal position to a different.

The complete game tree may be a game tree whose root is starting position, and every one the leaves are ending positions. Each complete game tree has as many nodes because the game has possible outcomes for each legal move made. it's easy to note that even for little games like tic-tac-toe the entire game tree is large . For that reason it's not an honest practice to explicitly create an entire game tree as a structure while writing a program that's alleged to predict the simplest move at any moment. Yet, the nodes should be created implicitly within the process of visiting.

We'll define state-space complexity of a game as variety of legal game positions reachable from the starting position of the game, and branching factor because the number of children at each node (if that number isn't constant, it is a common practice to use an average).

For tic-tac-toe, an upper bound for the size of the state space is 39=19683.

In strategic games, rather than letting the program start the searching process within the very beginning of the sport , an inventory of known and productive moves that are frequent and known to be productive while we still do not have much information about the state of game itself if we glance at the board.

In the beginning, it's too early within the game, and therefore the number of potential positions is just too great to automatically decide which move will definitely cause a far better game state (or win).

However, the algorithm reevaluates subsequent potential moves every turn, always choosing what at that moment appears to be the fastest route to victory. Therefore, it won't execute actions that take quite one move to finish , and is unable to perform certain documented "tricks" due to that. If the AI plays against a person's , it's very likely that human will immediately be ready to prevent this.

If, on the opposite hand, we take a glance at chess, we'll quickly realize the impracticality of solving chess by brute forcing through an entire game tree. To demonstrate this, Shannon calculated the boundary of the game-tree complexity of chess, resulting in about 10120 possible games.

Just how big is that number? For reference, if we compared the mass of an electron (10-30kg) to the mass of the entire known universe (1050-1060kg), the ratio would be in order of 1080-1090.

Let's take this instance to a tic-tac-toe game. As you almost certainly already know, the foremost famous strategy of player X is to start out in any of the corners, which provides the player O the foremost opportunities to form an error . If player O plays anything besides center and X continues his initial strategy, it is a guaranteed win for X.

**We are using Alpha-Beta pruning Algorithm for this game.**

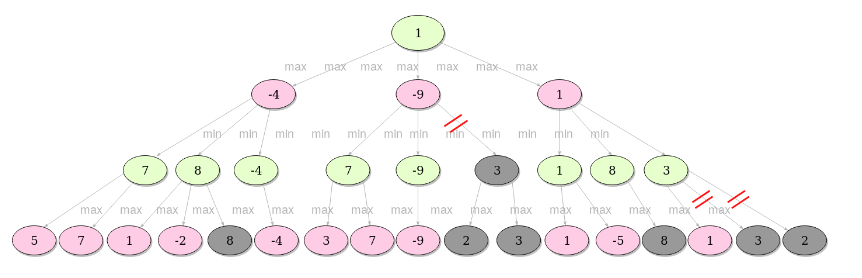
Alpha–beta is really an improved minimax employing a heuristic. It stops evaluating a move when it makes sure that it's worse than previously examined move. Such moves needn't to be evaluated further.

When added to an easy minimax algorithm, it gives an equivalent output, but cuts off certain branches that can't possibly affect the ultimate decision - dramatically improving the performance.

The main concept is to maintain two values through whole search:

* Alpha: Best already explored option for player Max
* Beta: Best already explored option for player Min

Initially, alpha is negative infinity and beta is positive infinity, i.e. in our code we'll be using the worst possible scores for both players.



When the search comes to the first grey area (8), it'll check the current best (with minimum value) already explored option along the path for the minimizer, which is at that moment 7. Since 8 is bigger than 7, we are allowed to cut off all the further children of the node we're at (in this case there aren't any), since if we play that move, the opponent will play a move with value 8, which is worse for us than any possible move the opponent could have made if we had made another move.

A better example may be when it comes to a next grey. Note the nodes with value -9. At that point, the best (with maximum value) explored option along the path for the maximizer is -4. Since -9 is less than -4, we are able to cut off all the other children of the node we're at.

This method allows us to ignore many branches that lead to values that won't be of any help for our decision, nor they would affect it in any way.

With that in mind, let's modify the min() and max() methods from before:

**PYTHON CODE:**

import random

def drawBoard(board):

# This function prints out the board that it was passed.

# "board" is a list of 10 strings representing the board (ignore index 0)

print(board[1] + '|' + board[2] + '|' + board[3])

print('-+-+-')

print(board[4] + '|' + board[5] + '|' + board[6])

print('-+-+-')

print(board[7] + '|' + board[8] + '|' + board[9])

def inputPlayerLetter():

# Lets the player type which letter they want to be.

# Returns a list with the player's letter as the first item, and the computer's letter as the second.

letter=''

while not(letter=='X' or letter=='O'):

print("Do you want to be 'X' or 'O'?")

letter = input().upper()

if letter == 'X':

return ['X','O']

else:

return ['O','X']

def whoGoesFirst():

print('Do you want to go first? (Yes or No)')

if input().lower().startswith('y'):

return 'player'

else:

return 'computer'

'''

# Randomly choose the player who goes first.

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

return 'computer'

else:

return 'player'

'''

def playAgain():

# This function returns True if the player wants to play again, otherwise it returns False.

print('Do you want to play again? (Yes or No)')

return input().lower().startswith('y')

def makeMove(board, letter, move):

board[move] = letter

def isWinner(board,letter):

# Given a board and a player's letter, this function returns True if that player has won.

return ((board[1]==letter and board[2]==letter and board[3]==letter) or

(board[4]==letter and board[5]==letter and board[6]==letter) or

(board[7]==letter and board[8]==letter and board[9]==letter) or

(board[1]==letter and board[4]==letter and board[7]==letter) or

(board[2]==letter and board[5]==letter and board[8]==letter) or

(board[3]==letter and board[6]==letter and board[9]==letter) or

(board[1]==letter and board[5]==letter and board[9]==letter) or

(board[3]==letter and board[5]==letter and board[7]==letter))

def getBoardCopy(board):

# Make a duplicate of the board list and return it the duplicate.

dupBoard = []

for i in board:

dupBoard.append(i)

return dupBoard

def isSpaceFree(board, move):

return board[move] == ' '

def getPlayerMove(board):

# Let the player type in their move.

move = ''

while move not in '1 2 3 4 5 6 7 8 9'.split() or not isSpaceFree(board,int(move)):

print('What is your next move? (1-9)')

move = input()

return int(move)

def chooseRandomMoveFromList(board, movesList):

# Returns a valid move from the passed list on the passed board.

# Returns None if there is no valid move.

possibleMoves = []

for i in movesList:

if isSpaceFree(board, i):

possibleMoves.append(i)

if len(possibleMoves) != 0:

return random.choice(possibleMoves)

else:

return None

def minimax(board, depth, isMax, alpha, beta, computerLetter):

# Given a board and the computer's letter, determine where to move and return that move.

if computerLetter == 'X':

playerLetter = 'O'

else:

playerLetter = 'X'

if isWinner(board, computerLetter):

return 10

if isWinner(board, playerLetter):

return -10

if isBoardFull(board):

return 0

if isMax:

best = -1000

for i in range(1,10):

if isSpaceFree(board, i):

board[i] = computerLetter

best = max(best, minimax(board, depth+1, not isMax, alpha, beta, computerLetter) - depth)

alpha = max(alpha, best)

board[i] = ' '

if alpha >= beta:

break

return best

else:

best = 1000

for i in range(1,10):

if isSpaceFree(board, i):

board[i] = playerLetter

best = min(best, minimax(board, depth+1, not isMax, alpha, beta, computerLetter) + depth)

beta = min(beta, best)

board[i] = ' '

if alpha >= beta:

break

return best

def findBestMove(board, computerLetter):

# Given a board and the computer's letter, determine where to move and return that move.

if computerLetter == 'X':

playerLetter = 'O'

else:

playerLetter = 'X'

bestVal = -1000

bestMove = -1

for i in range(1,10):

if isSpaceFree(board, i):

board[i] = computerLetter

moveVal = minimax(board, 0, False, -1000, 1000, computerLetter)

board[i] = ' '

if moveVal > bestVal:

bestMove = i

bestVal = moveVal

return bestMove

def isBoardFull(board):

# Return True if every space on the board has been taken. Otherwise return False.

for i in range(1,10):

if isSpaceFree(board, i):

return False

return True

print('\nWelcome to Tic Tac Toe!\n')

print('Reference of numbering on the board')

drawBoard('0 1 2 3 4 5 6 7 8 9'.split())

print('')

while True:

# Reset the board

theBoard = [' '] \* 10

playerLetter, computerLetter = inputPlayerLetter()

turn = whoGoesFirst()

print('The ' + turn + ' will go first.')

gameIsPlaying = True

while gameIsPlaying:

if turn == 'player':

drawBoard(theBoard)

move = getPlayerMove(theBoard)

makeMove(theBoard, playerLetter, move)

if isWinner(theBoard, playerLetter):

drawBoard(theBoard)

print('You won the game')

gameIsPlaying = False

else:

if isBoardFull(theBoard):

drawBoard(theBoard)

print('The game is a tie')

break

else:

turn = 'computer'

else:

move = findBestMove(theBoard, computerLetter)

makeMove(theBoard, computerLetter, move)

if isWinner(theBoard, computerLetter):

drawBoard(theBoard)

print('You lose the game')

gameIsPlaying = False

else:

if isBoardFull(theBoard):

drawBoard(theBoard)

print('The game is a tie')

break

else:

turn = 'player'

if not playAgain():

break