cp2410 aSSIGNMENT 2

Game Trees and Connect 3

Cameron Clark

JC260185

# Part 1 – Two player mode on a variable size board - Pseudocode:

## Two player mode, playgame.py

def run\_two\_player\_mode()

cols = ask player for number between 3, 7

rows = ask player for number between 3, 7

game = Connect3Board(cols, rows)

print game

# play until there is a winner or draw

While the game has no winner:

move = ask player for move between 0 and cols – 1

if move is a valid move

make move

print(updated board)

else:

print "That column is not available. Please choose again."

If the game is a draw:

Print draw game message

else:

print winning player message

## Program win detection, connect3board.py

### Planning the setup of any board

To access a node: [row] then [column]

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Row 0 | [0][0] | [0][1] | [0][2] | [0][3] | [0][4] |
| Row 1 | [1][0] |  |  |  |  |
| Row 2 | [2][0] |  |  |  |  |
| Row 3 | [3][0] |  |  |  |  |
| Columns: | 0 | 1 | 2 | 3 | 4 |

def get\_winner(self):

""" Returns None if the game is not complete, DRAW if no more moves can be played and there is no winner, or the token (O or #) that has won the game by making three-in-a-row horizontally, vertically, or diagonally."""

# Check rows for winner

For each row on board

For each col on board – 2

If board[row][col] is not none

if board[row][col] and board[row][col + 1] and board[row][col + 2] are the same piece

return board[row][col]

# Check columns for winner

for each col on board

for row on board

if board[row][col] is not none

if board[row][col] and board[row + 1][col] and board[row + 2][col] are the same piece:

return board[row][col]

# Check diagonal for winner (from top left)

for row on board - 2:

for col on board -2:

if board[row][col] is not None

if board[row][col] and board[row + 1][col + 1] and self.\_board[row+2][col+2] same piece:

return board[row][col]

# Check diagonal for winner (from bottom left)

for row on board - 2:

for col in range(2, self.\_cols):

if board[row][col] is not None

if board[row][col] and board[row + 1][col - 1] and\_board[row + 2][col – 2] are same piece:

return self.\_board[row][col]

# no winner discovered, so check for draw or otherwise return None

if turn\_number >= board rows \* board cols:

return Connect3Board.DRAW

else:

return None

# Part 2 - Game tree and minimax for AI - Pseudocode:

## Connect 3 game tree for 3x3 board with minimax scoring, gametree.py

class GameTree:  
 MAX\_PLAYER = 'O'  
 MIN\_PLAYER = '#'  
 MAX\_WIN\_SCORE = 1  
 MIN\_WIN\_SCORE = -1  
 DRAW\_SCORE = 0  
  
 class \_Node:  
  
 def \_\_init\_\_(self, gameboard: Connect3Board):  
 self.\_gameboard = gameboard  
 self.\_children = [None] \* self.\_gameboard.get\_columns()  
  
 if gameboard winner is None:  
 self.\_create\_children()  
 self.\_compute\_score()  
 elif gameboard winner == gameboard.DRAW:  
 self.\_score = GameTree.DRAW\_SCORE  
 elif gameboard winner == GameTree.MAX\_PLAYER:  
 self.\_score = GameTree.MAX\_WIN\_SCORE  
 elif gameboard winner == GameTree.MIN\_PLAYER:  
 self.\_score = GameTree.MIN\_WIN\_SCORE  
  
 def \_create\_children(self):

for each column on board

if move is possible on column

make a copy of board as board\_copy

add the move on board\_copy on column

self.\_children[col] = GameTree.\_Node(board\_copy)  
  
 def \_compute\_score(self):

if it is GameTree.MAX\_PLAYER’s turn

max\_score = -1

for each child in self.\_children

if child is not None and child’s score > max\_score

max\_score = child’s score

self.\_score = max\_score  
 else:  
 min\_score = 1  
 for each child in self.\_children:  
 if child is not None and child’s score < min\_score:  
 min\_score = child’s score  
 self.\_score = min\_score

## AI opponent mode of Connect 3 on a 3×3 board, playgame.py

Def run\_ai\_mode():

player let player choose piece (O will always go first)

cols = 3  
 rows = 3  
 game = Connect3Board(cols, rows)  
 print(game)  
  
 game\_tree = GameTree(game)  
 position = game\_tree.get\_root\_position()  
  
 while game winner is None:  
 if game.get\_whose\_turn() == player:  
 move = ask player for valid move between 0 and cols – 1  
 if move is valid:  
 game.add\_token(move)  
 position = position.get\_child(move)  
 print(game)  
 else:  
 print("ERROR: Invalid move, please try again")  
 else:  
 children\_scores = position.get\_children\_scores()  
 child\_index = None  
 max\_score = -1  
 min\_score = 1

index = 0

for each child in children\_scores:  
 if it is O turn:  
 if child is not None and child > max\_score:  
 max\_score = child  
 child\_index = index  
 else:  
 if child is not None and child < min\_score:  
 min\_score = child  
 child\_index = index

index += 1  
  
 game.add\_token(child\_index)  
 position = position.get\_child(child\_index)  
  
 print("AI's turn")  
 print(game)  
  
 if game is draw:  
 print draw message  
 else:  
 print which player has won

# Part 3 – Analysis

## Big O analysis of running time of get\_winner

Let

r = number of rows

c = number of columns

r \* (c-2) + c \* (r-2) + (r–2) \* (c-2) + (r–2) \* (c-2)

Simplified into

4cr -6c – 6r + 8

Which leaves:

Big O as cr

Code for reference

**def get\_winner**(self):  
 *""" Returns None if the game is not complete, DRAW if no more moves can be played and there is no winner,  
 or the token (O or #) that has won the game by making three-in-a-row horizontally, vertically, or diagonally."""* # Check rows for winner  
 **for** row **in** range(self.\_rows):  
 **for** col **in** range(self.\_cols - 2):  
 **if** self.\_board[row][col] **is not None and** (self.\_board[row][col] == self.\_board[row][col + 1] == self.\_board[row][col + 2]):  
 **return** self.\_board[row][col]  
  
 # Check columns for winner  
 **for** col **in** range(self.\_cols):  
 **for** row **in** range(self.\_rows - 2):  
 **if** self.\_board[row][col] **is not None and** (self.\_board[row][col] == self.\_board[row + 1][col] == self.\_board[row + 2][col]):  
 **return** self.\_board[row][col]  
  
 # Check diagonal for winner (from top left)  
 **for** row **in** range(self.\_rows - 2):  
 **for** col **in** range(self.\_cols - 2):  
 **if** self.\_board[row][col] **is not None and** (self.\_board[row][col] == self.\_board[row + 1][col + 1] == self.\_board[row + 2][col + 2]):  
 **return** self.\_board[row][col]  
  
 # Check diagonal for winner (from bottom left)  
 **for** row **in** range(self.\_rows - 2):  
 **for** col **in** range(2, self.\_cols):  
 **if** self.\_board[row][col] **is not None and** (self.\_board[row][col] == self.\_board[row + 1][col - 1] == self.\_board[row + 2][col - 2]):  
 **return** self.\_board[row][col]  
  
 # no winner discovered, so check for draw or otherwise return None  
 **if** self.\_turn\_number >= self.\_rows \* self.\_cols:  
 **return** Connect3Board.DRAW  
 **else**:  
 **return None**

## Calculating upper bound

Note: This is a larger number than the actual size of the game tree, as some nodes of course end early because the game has been won by one of the players. A full correct analysis wouldn't admit a simple solution.

Feasible storage would mean could fit in main memory on a reasonable computer:

Assume 16GB

Assume longest game will last is r x c moves

Assume each node requires about r x c bytes of storage

at each stage the tree branches out c more nodes

Therefore storage = rcrc bytes

1 gigabyte = 1e+9 bytes

16 gigabyte = 1.6e+10 bytes

If there are:

4 rows

4 col

Storage = rcrc bytes

= 16^16

= 1.84e+19 (too large)

If there are:

4 rows

3 col

Storage = rcrc bytes

= 12^12

= 8.9e+12

If there are:

3 rows

3 col

Storage = rcrc bytes

= 9^9

= 3.87e+8 < 1.6e+10 bytes

Therefore, largest board with this model is 3x3. Again, this is wildly inaccurate as it factors invalid moves and bloats the game tree.