**COP 4630**

**DR MARQUES**

**Summer 2022**

***Armando Arias Developer***

***Joshua Garden Architect***

***Flor Aguirre Reporter***

Github: https://github.com/faguirregarc2015/GroupAI

|  |  |
| --- | --- |
|  | REPORT PROJECT I  Game Implementation using the easyAI Framework |
|  |  |
|  |  |

**BUILDING GAMES WITH ARTIFICIAL INTELLIGENCE**

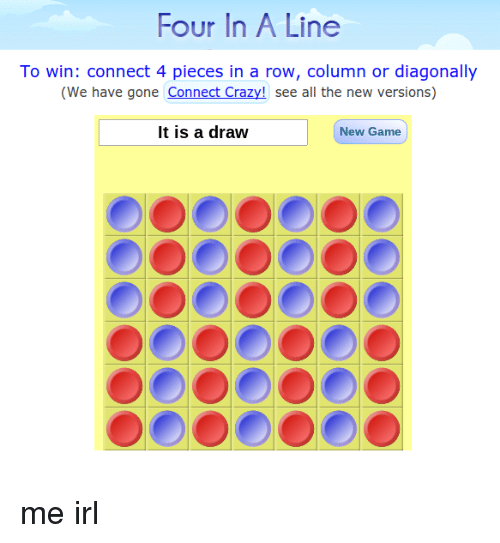
**Purpose:**

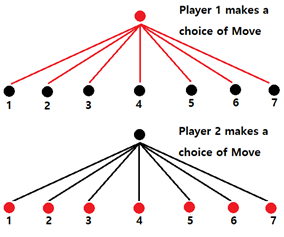
The purpose of this Assignment is to build games using search algorithms to effectively come up with strategies to win games, including Tic Tac Toe, Connector four and checkers with artificial intelligence, through the use of Searching algorithms, in this case we will use the MinMax Search.

**Theory**:

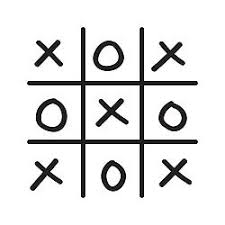
In Artificial Intelligence, games are deterministic, using turns as two players or zero-sum, which describe if a participant will win or loss. A game with two players is a type of adversarial search such as min -max useful when something else affects the environment, where MAX moves first, and MIN is the response to that move.

Adversarial Search is characterized by opposition, it requires the anticipation and understanding of the actions of the opponent in pursuit of a goal, this is the type of search utilized in games like tic-tac-toe, connect Four and checkers.

**Connect Four** is a two-player game, **is an example of an adversarial, finite zero-sum game of perfect information**. This means that there are two players, against each other in a game where only one player can win. In this game the players take turns dropping the tokens into a vertical grid, the goal is to get for tokens in a line.

 Text

Description automatically generated

Connect Four Diagram: https://medium.com/analytics-vidhya/artificial-intelligence-at-play-connect-four-minimax-algorithm-explained-3b5fc32e4a4f

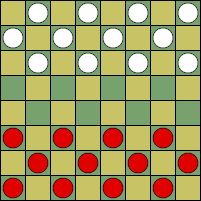
**Tic-Tac-Toe**: The idea is that by anticipating the moves of the opponent in response to moves by the algorithm, the algorithm can come up with the best possible next move.

Graphical user interface, application, Word

Description automatically generatedDiagram, engineering drawing

Description automatically generated

Tic tac toe Diagram: A partial decision tree for the game of tic-tac-toe taken from **Artificial Intelligence: A Modern Approach By Stuart Russel and Peter Norvig** page 163.

**Checkers**: The objective of the game is to get as many pieces as possible from the opponent, the game can be won when the opponent is unable to make a move, because all the pieces were captured, or the opponent get the king.

Pseudocode:

(TwoPlayerGame)

Int(self, players, board){

//Set a 8\*8 board

W[iI,j]=8

Assign: players[0] = white pieces

Players[1] = black pieces}

MinMax (Negamax, self, Pos){

Return MaxMove (self);

}

MaxMove (Negamax, self, pos)

If(PossibleMoves(pos) {

//Result

Return True: Self.black\_territory = 0 || self.white\_territory=1}

Else {

Move < - {}

Move <- Generate\_move(game)

For each move {

Move <- MinMove(self.position);

If (value(ove) > value (best\_move)) {

Best\_move <- move; }}

Return best\_move: }

MinMove(self, pos)

Best\_move <- {};

Moves <- GenerateMoves(game)

For each moves {

Move <- MaxMove(selfposition);

If (value(move) > value(best\_move)) {

Best\_move <- move;}}

Return best\_move: }

**How to Build the Tree**:

* Start: From the root of the tree.
  + Node has several children (represented by the moves)
    - More children that represent the states after the opponent moves.
  + Result of the game (after executing various moves)
* End: When one of the players win.

**The sequence of tree processing using min-max search**

Diagram

Description automatically generated

**Flow of Min-Max Search Algorithm**:

**Conclusions**:

Min-Max is type of adversarial of intelligence search, the goal is to maximize the benefit for a player, we call it adversarial because in games like Tic tac toe, checkers, and connect four, there is two players that have the goal to counteract the actions of the opponent and maximize the benefit for the player, in this type of the search the authors of the game take turns for the opportunity to change the state of the environment of the game to their favor.

How it works?

Heuristic: AI that take decision sbased on rules to get a score.

The goal is to find the best possible move for a player. This can be done by choosing the node with the best evaluation score. The best choice will be made after evaluating all the potential moves of the opponent. The algorithm looks ahead at all the possible values till the end and makes a decision for the player, finally present a score where select the winner of the game.

**IMPLEMENTATION REPORT**

**Part I:**

1. Conda Installation and Configuration

Text

Description automatically generated

A screenshot of a computer

Description automatically generated

1. EasyAI Installation

Text

Description automatically generated

**Parte 2: Run Tic-Tac-Toe presented in easyAI documentation**

**Code:**

from easyAI import TwoPlayerGame

from easyAI.Player import Human\_Player

class TicTacToe( TwoPlayerGame ):

""" The board positions are numbered as follows:

1 2 3

4 5 6

7 8 9

"""

def \_\_init\_\_(self, players):

self.players = players

self.board = [0]\*9

self.current\_player = 1 # player 1 starts.

def possible\_moves(self):

return [i+1 for i,e in enumerate(self.board) if e==0]

def make\_move(self, move):

self.board[int(move)-1] = self.current\_player

def unmake\_move(self, move): # optional method (speeds up the AI)

self.board[int(move)-1] = 0

def lose(self):

""" Has the opponent "three in line ?" """

return any( [all([(self.board[c-1]== self.current\_player)

for c in line])

for line in [[1,2,3],[4,5,6],[7,8,9], # horiz.

[1,4,7],[2,5,8],[3,6,9], # vertical

[1,5,9],[3,5,7]]]) # diagonal

def is\_over(self):

return (self.possible\_moves() == []) or self.lose()

def show(self):

print ('\n'+'\n'.join([

' '.join([['.','O','X'][self.board[3\*j+i]]

for i in range(3)])

for j in range(3)]) )

def scoring(self):

return -100 if self.lose() else 0

if \_\_name\_\_ == "\_\_main\_\_":

from easyAI import AI\_Player, Negamax

ai\_algo = Negamax(6)

TicTacToe( [Human\_Player(),AI\_Player(ai\_algo)]).play()

Graphical user interface, application, Word

Description automatically generatedGraphical user interface, application, Word

Description automatically generated

**Four Connect Game:**

**Code**

#!/usr/bin/env python3

try:

import numpy as np

except ImportError:

print("Sorry, this example requires Numpy installed !")

raise

from easyAI import TwoPlayerGame

class ConnectFour(TwoPlayerGame):

"""

The game of Connect Four, as described here:

http://en.wikipedia.org/wiki/Connect\_Four

"""

def \_\_init\_\_(self, players, board = None):

self.players = players

self.board = board if (board != None) else (

np.array([[0 for i in range(7)] for j in range(6)]))

# self.nplayer = 1 # player 1 starts.

self.current\_player = 1

def possible\_moves(self):

return [i for i in range(7) if (self.board[:, i].min() == 0)]

def make\_move(self, column):

line = np.argmin(self.board[:, column] != 0)

self.board[line, column] = self.current\_player

def show(self):

print('\n' + '\n'.join(

['0 1 2 3 4 5 6', 13 \* '-'] +

[' '.join([['.', 'O', 'X'][self.board[5 - j][i]]

for i in range(7)]) for j in range(6)]))

def lose(self):

return find\_four(self.board, self.current\_player)

def is\_over(self):

return (self.board.min() > 0) or self.lose()

def scoring(self):

return -100 if self.lose() else 0

def find\_four(board, nplayer):

"""

Returns True iff the player has connected 4 (or more)

This is much faster if written in C or Cython

"""

for pos, direction in POS\_DIR:

streak = 0

while (0 <= pos[0] <= 5) and (0 <= pos[1] <= 6):

if board[pos[0], pos[1]] == nplayer:

streak += 1

if streak == 4:

return True

else:

streak = 0

pos = pos + direction

return False

POS\_DIR = np.array([[[i, 0], [0, 1]] for i in range(6)] +

[[[0, i], [1, 0]] for i in range(7)] +

[[[i, 0], [1, 1]] for i in range(1, 3)] +

[[[0, i], [1, 1]] for i in range(4)] +

[[[i, 6], [1, -1]] for i in range(1, 3)] +

[[[0, i], [1, -1]] for i in range(3, 7)])

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

# LET'S PLAY !

from easyAI import Human\_Player, AI\_Player, Negamax, SSS, DUAL

ai\_algo\_neg = Negamax(5)

ai\_algo\_sss = SSS(5)

game = ConnectFour([AI\_Player(ai\_algo\_neg), AI\_Player(ai\_algo\_sss)])

game.play()

if game.lose():

print("Player %d wins." % (game.current\_player))

else:

print("Looks like we have a draw.")

A screenshot of a computer

Description automatically generated with medium confidence

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

**Checkers Code:**

#Project 1 CAP4630 Summer 2022

#Professor: Oge Marques PhD

#Topic: Checkers game

#Updated: 6-13-2022

#Updated by: Armando Arias-Castro

#Created by: Anak Wannaphaschaiyong

#Team members:

#Flor Aguirre Garcia - Documentation

#Joshua Gardner - Designer

#Armando Arias-Castro - Developer

#Github: https://github.com/faguirregarc2015/GroupAI

from easyAI import TwoPlayerGame, Human\_Player, AI\_Player, Negamax

from easyAI import solve\_with\_iterative\_deepening

import numpy as np

# black\_square

even = [0,2,4,6]

odd = [1,3,5,7]

# init

even\_row = [(i,j) for i in even for j in odd]

odd\_row = [(i,j) for i in odd for j in even]

black\_squares = even\_row + odd\_row

class Checker(TwoPlayerGame):

def \_\_init\_\_(self, players):

self.players = players

# self.board = np.arange(8 \* 8).reshape(8,8)

self.blank\_board = np.zeros((8,8), dtype=object)

self.board = self.blank\_board.copy()

self.black\_pieces = [

(0,1), (0,3), (0,5), (0,7),

(1,0), (1,2), (1,4), (1,6)

]

self.white\_pieces = [

(6,1), (6,3), (6,5), (6,7),

(7,0), (7,2), (7,4), (7,6)

]

for i,j in self.black\_pieces:

self.board[i,j] = "B"

for i,j in self.white\_pieces:

self.board[i,j] = "W"

self.white\_territory = [(7,0), (7,2), (7,4), (7,6)]

self.black\_territory = [(0,1), (0,3), (0,5), (0,7)]

self.players[0].pos = self.white\_pieces

self.players[1].pos = self.black\_pieces

self.current\_player = 1 # player 1 starts.

def possible\_moves\_on\_white\_turn(self):

table\_pos = []

old\_new\_piece\_pos = []

# board position before move

board = self.blank\_board.copy()

for (p,l) in zip(self.players, ["W", "B"]):

for x,y in p.pos:

board[x,y] = l

# get legal move of each pieces. (old piece location, new piece location)

# get position of each move (list of all table position)

for v in self.players[self.current\_player-1].pos:

old\_piece\_pos = v

step\_pos = [(v[0]-1, v[1]-1), (v[0]-1, v[1]+1)]

# if no piece at step\_pos, step

# otherwise jump until no piece at next step\_pos

for n in step\_pos:

if (n[0] >= 0 and n[0] <= 7) and (n[1] >= 0 and n[1] <= 7) and (n in black\_squares):

if board[n[0], n[1]] in ["B","W"]:

y = ((n[0] - old\_piece\_pos[0]) \* 2) + old\_piece\_pos[0]

x = ((n[1] - old\_piece\_pos[1]) \* 2) + old\_piece\_pos[1]

j = (y,x)

is\_inside\_board = (j[0] >= 0 and j[0] <= 7) and (j[1] >= 0 and j[1] <= 7)

if (j[0] <= 7) and (j[1] <=7):

is\_position\_empty = (board[j[0], j[1]] == 0)

else:

is\_position\_empty = False

if is\_inside\_board and (j in black\_squares) and is\_position\_empty:

# print(old\_piece\_pos,j)

old\_new\_piece\_pos.append((old\_piece\_pos,j))

else:

old\_new\_piece\_pos.append((old\_piece\_pos,n))

# board position after move

for i,j in old\_new\_piece\_pos:

#print(f"i = {i}")

b = board.copy()

b[i[0], i[1]] = 0 # old position

b[j[0], j[1]] = "W" # new position

# print(b)

table\_pos.append(b)

assert len(np.where(b != 0)[0]) == 16, f"In possible\_moves\_on\_white\_turn(), there are {len(np.where(b != 0)[0])} pieces on the board \n {b}"

self.board = board

return table\_pos

def possible\_moves\_on\_black\_turn(self):

table\_pos = []

old\_new\_piece\_pos = []

# board position before move

board = self.blank\_board.copy()

for (p,l) in zip(self.players, ["W", "B"]):

for x,y in p.pos:

board[x,y] = l

# get legal move of each pieces. (old piece location, new piece location)

# get position of each move (list of all table position)

for v in self.players[self.current\_player-1].pos:

old\_piece\_pos = v

step\_pos = [(v[0]+1, v[1]-1), (v[0]+1, v[1]+1)]

# if no piece at step\_pos, step

# otherwise jump until no piece at next step\_pos

for n in step\_pos:

if (n[0] >= 0 and n[0] <= 7) and (n[1] >= 0 and n[1] <= 7) and (n in black\_squares):

if board[n[0], n[1]] in ["B","W"]:

y = ((n[0] - old\_piece\_pos[0]) \* 2) + old\_piece\_pos[0]

x = ((n[1] - old\_piece\_pos[1]) \* 2) + old\_piece\_pos[1]

j = (y,x)

is\_inside\_board = (j[0] >= 0 and j[0] <= 7) and (j[1] >= 0 and j[1] <= 7)

if (j[0] <= 7) and (j[1] <=7):

is\_position\_empty = (board[j[0], j[1]] == 0)

else:

is\_position\_empty = False

if is\_inside\_board and (j in black\_squares) and is\_position\_empty:

# print(old\_piece\_pos,j)

old\_new\_piece\_pos.append((old\_piece\_pos,j))

else:

old\_new\_piece\_pos.append((old\_piece\_pos,n))

# board position after move

for i,j in old\_new\_piece\_pos:

b = board.copy()

b[i[0], i[1]] = 0

b[j[0], j[1]] = "B"

table\_pos.append(b)

assert len(np.where(b != 0)[0]) == 16, f"In possible\_moves\_on\_black\_turn(), there are {len(np.where(b != 0)[0])} pieces on the board \n {b}"

self.board = board

return table\_pos

def possible\_moves(self):

if self.current\_player == 2:

return self.possible\_moves\_on\_black\_turn()

else:

return self.possible\_moves\_on\_white\_turn()

print("In possible moves")

def get\_piece\_pos\_from\_table(self, table\_pos):

if self.current\_player-1 == 0:

x = np.where(table\_pos == "W")

elif self.current\_player-1 == 1:

x = np.where(table\_pos == "B")

else:

raise ValueError("There can be at most 2 players.")

assert len(np.where(table\_pos != 0)[0]) == 16, f"In get\_piece\_pos\_from\_table(), there are {len(np.where(table\_pos != 0)[0])} pieces on the board \n {table\_pos}"

return [(i,j) for i,j in zip(x[0], x[1])]

***def make\_move(self, pos):***

***"""***

***This function receives as parameter variable pos.***

***Variable pos contains a board that shows a possible move.***

***The function will extract a list with the position of all the pieces***

***from pos using self.get\_piece\_pos\_from\_table(pos).***

***After checking which player's turn it is to play,***

***the list is assigned to them.***

***Finally the board is updated using variable pos.***

***The function returns the updated board***

***"""***

***#Create a copy of the position array passed as parameter***

***#It will be used to update the board at the end***

***new\_move = pos.copy()***

***#Create a list with all the positions of the pieces***

***#in the array pos passed as parameter***

***lst = self.get\_piece\_pos\_from\_table(pos)***

***#Check which player's turn it is to play***

***#Player 0 is white pieces***

***#Player 1 is black pieces***

***#Assigned the list of position to the position array***

***#of the corresponding player***

***if self.current\_player-1 == 0:***

***self.players[0].pos = lst***

***elif self.current\_player-1 == 1:***

***self.players[1].pos = lst***

***#Update the board***

***self.board = new\_move.copy()***

***#Return the updated board***

***return self.board***

***def lose(self):***

***"""***

***Black lose if white piece is in black territory***

***White lose if black piece is in white territory***

***This function verifies if any white pieces are in black territory***

***or any black pieces are in white territory.***

***The function also checks if there are no possible moves.***

***The function will return True or False.***

***"""***

***#Check if there are any possible moves***

***if self.possible\_moves() == []:***

***return True***

***#Check if any white pieces are in black territory***

***for i in self.black\_territory:***

***if i in self.players[0].pos:***

***return True***

***#Check if any black pieces are in white territory***

***for i in self.white\_territory:***

***if i in self.players[1].pos:***

***return True***

***#Return False if no player has won and there are possible moves***

***return False***

***def is\_over(self):***

***"""***

***Game is over immediately when one player get one of its piece into opponent's territory.***

***This function calls function self.lose and return the same result.***

***The function will return True or False.***

***"""***

***return self.lose()***

***def show(self):***

***"""***

***show 8\*8 checker board.***

***"""***

***board = self.blank\_board.copy()***

***print(f"player 1 positions = {self.players[0].pos}")***

***print(f"player 2 positions = {self.players[1].pos}")***

***for (p,l) in zip(self.players, ["W", "B"]):***

***for x,y in p.pos:***

***board[x,y] = l***

***print('\n')***

***print(board)***

***self.is\_over()#Check if any player won already***

***def scoring(self):***

***"""***

***win = 0***

***lose = -100***

***This function returns 0 if the player won and -100 if the player lost***

***This function calls function self.lose to check wether a player won or not***

***"""***

***return -100 if self.lose() else 0***

if \_\_name\_\_ == "\_\_main\_\_":

ai = Negamax(1) # The AI will think 13 moves in advance

game = Checker( [ AI\_Player(ai), AI\_Player(ai) ] )

history = game.play()

Graphical user interface, application

Description automatically generatedGraphical user interface, application

Description automatically generated

Graphical user interface, application

Description automatically generatedGraphical user interface, application, Word

Description automatically generated

Graphical user interface, application

Description automatically generatedGraphical user interface, application

Description automatically generated

**Output from test\_checkers.py**

