**# Assignment4.py**

**# Lizzie Siegle and Sujin Kay**

**# Implements a program to play the game of Konane (Hawaiian Checkers).**

import random

from copy import deepcopy

"""

CREATES THE NODE CLASS (to be used to represent various game states)

"""

class Node:

def \_\_init\_\_(self, board, move, level, depth\_limit, player, who\_first):

"Instance variables."

self.board = board # current board/game state

self.move = move # move that brought you to this state

self.level = level # node at level L in the search tree

self.depth\_limit = depth\_limit # depth limit for searching

self.player = player # which player goes next

self.who\_first = who\_first # who goes first in the game

"""

BOARD CLASS and FOR BOARD/GAME FUNCTIONALITY

"""

class BOARD:

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

self.width = width

self.board = [[' ']\*(self.width + 1) for row in range(self.width + 1)]

"Creates the 8x8 board display, using X for dark pieces and O for light pieces."

def create\_board(self, width):

for row in range(self.width + 1):

for col in range(self.width + 1):

# Set board's horizontal and vertical coordinate lines."

if (row == 0):

self.board[row][col] = col

self.board[row][0] = ' ' # replace coordinate in (0,0) with a blank space

elif (col == 0):

self.board[row][col] = row

# Set board's alternating X's and O's.

elif ((row + col) % 2 == 0):

self.board[row][col] = 'X'

else:

self.board[row][col] = 'O'

return self.board

"Prints the board."

def print\_board(self, board):

for row in range(self.width + 1):

for col in range(self.width + 1):

print (board[row][col]), # print on one line

print

print

"""

FOR GENERATING POSSIBLE MOVES

"""

"Generates all possible first moves."

def possible\_first\_moves(self):

first\_moves = []

first\_moves.append((1, 1)) # top left corner

first\_moves.append((self.width/2, self.width/2)) # middle piece on left side

first\_moves.append((self.width/2+1, self.width/2+1)) # middle piece on right side

first\_moves.append((self.width, self.width)) # bottom right corner

return first\_moves

"Generates all possible second moves."

def possible\_second\_moves(self, first\_move):

second\_moves = []

# if first move removed top left piece

if (self.board[1][1] == ' '):

second\_moves.append((1, 2))

second\_moves.append((2, 1))

return second\_moves

# if first move removed bottom right piece

elif (self.board[self.width][self.width] == ' '):

second\_moves.append((self.width-1, self.width))

second\_moves.append((self.width, self.width-1))

return second\_moves

# if first move removed one of the middle pieces

else:

second\_moves.append((first\_move[0]+1, first\_move[1]))

second\_moves.append((first\_move[0]-1, first\_move[1]))

second\_moves.append((first\_move[0], first\_move[1]+1))

second\_moves.append((first\_move[0], first\_move[1]-1))

return second\_moves

"Generates all possible moves, INCLUDING multiple jumps."

def generate\_moves(self, board, X\_or\_O):

possible\_moves = []

jump\_to = (0, 0)

up = down = left = right = 2

for row in range(self.width + 1):

for col in range(self.width + 1):

"Specify whether looking for Dark/Light moves."

if (board[row][col] == X\_or\_O):

"Can this piece move North?"

# current position

current\_pos = (row, col)

# is move within scope of the board?

while ((current\_pos[0] - up) > 0):

jump\_to = (current\_pos[0] - up, col)

# is there a blank space where we need to jump, and is there an opponent's piece to jump over?

if (board[jump\_to[0]][jump\_to[1]] == ' ' and board[jump\_to[0]+1][jump\_to[1]] != ' '):

# if so, then append this move

possible\_moves.append((row, col, jump\_to[0], jump\_to[1]))

# update how far you'll jump next time (for multiple jumps)

current\_pos = jump\_to

else:

break

"Can this piece move South?"

# current position

current\_pos = (row, col)

# is move within scope of the board?

while ((current\_pos[0] + down) < self.width+1):

jump\_to = (current\_pos[0] + down, col)

# is there a blank space where we need to jump, and is there an opponent's piece to jump over?

if (board[jump\_to[0]][jump\_to[1]] == ' ' and board[jump\_to[0]-1][jump\_to[1]] != ' '):

# if so, then append this move

possible\_moves.append((row, col, jump\_to[0], jump\_to[1]))

# update how far you'll jump next time (for multiple jumps)

current\_pos = jump\_to

else:

break

"Can this piece move East?"

# current position

current\_pos = (row, col)

# is move within scope of the board?

while ((current\_pos[1] + right) < self.width+1):

jump\_to = (row, current\_pos[1] + right)

# is there a blank space where we need to jump, and is there an opponent's piece to jump over?

if (board[jump\_to[0]][jump\_to[1]] == ' ' and board[jump\_to[0]][jump\_to[1]-1] != ' '):

# if so, then append this move

possible\_moves.append((row, col, jump\_to[0], jump\_to[1]))

# update how far you'll jump next time (for multiple jumps)

current\_pos = jump\_to

else:

break

"Can this piece move West?"

# current position

current\_pos = (row, col)

# is move within scope of the board?

while ((current\_pos[1] - left) > 0):

jump\_to = (row, current\_pos[1] - left)

# is there a blank space where we need to jump, and is there an opponent's piece to jump over?

if (board[jump\_to[0]][jump\_to[1]] == ' ' and board[jump\_to[0]][jump\_to[1]+1] != ' '):

# if so, then append this move

possible\_moves.append((row, col, jump\_to[0], jump\_to[1]))

# update how far you'll jump next time (for multiple jumps)

current\_pos = jump\_to

else:

break

return possible\_moves

"""

SPECIAL CASES: FIRST AND SECOND MOVES

"""

def first\_second\_move(self, user\_turn):

first\_moves = self.possible\_first\_moves()

# if the user goes first...

if (user\_turn):

# give instructions

print "For the first move, only " + str(first\_moves[0]) + ", " + str(first\_moves[1]) + ", " + str(first\_moves[2]) + ", or " + str(first\_moves[3]) + " allowed."

# ask for user input

coord = input("Enter your move, in the form (x, y): ")

# make sure that input is valid

while (coord not in first\_moves):

print "Error -- invalid move. Please try again."

coord = input("Enter your move, in the form (x, y): ")

# remove this piece from the board

print "User removes " + str(coord)

self.board[coord[0]][coord[1]] = ' '

self.print\_board(self.board)

# computer goes second, chooses a random legal move

second\_moves = self.possible\_second\_moves(coord)

coord2 = random.choice(second\_moves)

# remove this piece from the board

print "Computer removes " + str(coord2)

self.board[coord2[0]][coord2[1]] = ' '

self.print\_board(self.board)

return self.board

# if the computer goes first...

else:

# choose a random legal move

coord = random.choice(first\_moves)

# remove this piece from the board

print "Computer removes " + str(coord)

self.board[coord[0]][coord[1]] = ' '

self.print\_board(self.board)

# give user instructions

second\_moves = self.possible\_second\_moves(coord)

print "For the second move, can only remove a piece ajacent to first move."

# ask for user input

coord2 = input("Enter your move, in the form (x, y): ")

# make sure that input is valid

while (coord2 not in second\_moves):

print "Error -- invalid move. Please try again."

coord2 = input("Enter your move, in the form (x, y): ")

# remove this piece from the board

print "User removes " + str(coord2)

self.board[coord2[0]][coord2[1]] = ' '

self.print\_board(self.board)

return self.board

"""

FOR GETTING AND MAKING MOVES

"""

"Asks the User for their move, or decided the best move for the Computer."

def get\_move(self, user\_turn, possible\_moves, who\_first):

# if it's the user's move...

if (user\_turn):

# ask for user input -- needs TWO coordinates

coordinates = input("Enter your move, in the form (x, y, x2, y2): ")

# make sure that input is valid

while (coordinates not in possible\_moves):

print "Error -- invalid move. Please try again."

coordinates = input("Enter your move, in the form (x, y, x2, y2): ")

return coordinates

# if it's the computer's move...

else:

"RandomPlayer: chooses a random move out of possible legal moves."

#best\_move = random.choice(possible\_moves)

"SmartPlayer: chooses the best move using minimax and alphabeta pruning."

if who\_first == 'User':

first\_node = Node(self.board, None, 0, 4, 'O', who\_first) # 4 is the depth limit

else:

first\_node = Node(self.board, None, 0, 4, 'X', who\_first)

bv\_move = self.minimax\_alpha\_beta(first\_node, float('-inf'), float('inf'))

best\_move = bv\_move[1]

return best\_move

"Plays the move on the game board."

def make\_move(self, move, X\_or\_O, to\_print):

# only print if moves are actually being made on game board (not for creating game boards for successor moves)

if (to\_print):

if (X\_or\_O == 'X'):

print "Dark moves (" + str(move[0]) + ", " + str(move[1]) + ") to (" + str(move[2]) + ", " + str(move[3]) + ")"

else:

print "Light moves (" + str(move[0]) + ", " + str(move[1]) + ") to (" + str(move[2]) + ", " + str(move[3]) + ")"

# if the move is horizontal (coordinates are in the same row)

if (move[0] == move[2]):

# remove your original piece

self.board[move[0]][move[1]] = ' '

# each time you jump, remove the opponent's piece that youre jumping over

current\_col = move[1]

# if you're going East...

if (move[3] > move[1]):

while (current\_col < move[3]):

# remove the opponent's piece from the board

self.board[move[0]][current\_col+1] = ' '

current\_col += 2

# if you're going West...

else:

while (current\_col > move[3]):

# remove the opponent's piece from the board

self.board[move[0]][current\_col-1] = ' '

current\_col -= 2

# insert your jumping piece to the final spot

self.board[move[2]][move[3]] = X\_or\_O

if (to\_print == True):

self.print\_board(self.board)

return self.board

# if the move is vertical (coordinates are in the same column)

else:

# remove your original piece

self.board[move[0]][move[1]] = ' '

# each time you jump, remove the opponent's piece that youre jumping over

current\_row = move[0]

# if you're going North...

if (move[0] > move[2]):

while (current\_row > move[2]):

# remove the opponent's piece from the board

self.board[current\_row-1][move[1]] = ' '

current\_row -= 2

# if you're going South...

else :

while (current\_row < move[2]):

# remove the opponent's piece from the board

self.board[current\_row+1][move[1]] = ' '

current\_row += 2

self.board[move[2]][move[3]] = X\_or\_O

if (to\_print == True):

self.print\_board(self.board)

return self.board

"""

MINIMAX AND STATIC EVALUATIONS

"""

"Generates all successor nodes for a current board game state."

def generate\_successor\_nodes(self, board, current\_node):

successor\_nodes = []

# generate all possible moves for the opponent

possible\_successor\_moves = self.generate\_moves(board, current\_node.player)

# for each possible move...

for move in possible\_successor\_moves:

# create a copy of the current game board

current\_state = deepcopy(self)

# make the move on the copy of the board

current\_state.make\_move(move, current\_node.player, False)

# create new node containing this new board, depending on who is the next player

if current\_node.player == 'X':

successor\_node = Node(current\_state.board, move, current\_node.level+1, current\_node.depth\_limit, 'O', current\_node.who\_first)

successor\_nodes.append(successor\_node)

else:

successor\_node = Node(current\_state.board, move, current\_node.level+1, current\_node.depth\_limit, 'X', current\_node.who\_first)

successor\_nodes.append(successor\_node)

return successor\_nodes

"Static evaluation function for possible moves (#our\_moves - #opponent's\_moves)."

def static\_eval(self, node):

score = 0

# figure out which piece corresponds to which player

if node.who\_first == 'User':

user\_piece = 'X'

computer\_piece = 'O'

else:

user\_piece = 'O'

computer\_piece = 'X'

# count the number of moves the computer can make

successor\_moves = self.generate\_moves(node.board, computer\_piece)

num\_moves = len(successor\_moves)

# count the number of moves the user can make

opp\_successor\_moves = self.generate\_moves(node.board, user\_piece)

num\_opponent\_moves = len(opp\_successor\_moves)

# calculate score (computer's possible moves - user's possible moves)

score = num\_moves - num\_opponent\_moves

return score

"Minimax -- returns the best move as defined by the static evaluation function."

def minimax(self, node):

# if node is at depth limit...

if (node.level == node.depth\_limit):

# do a static evaluation, return result and the best move

return (self.static\_eval(node), node.move)

# generate successor nodes

successor\_nodes = self.generate\_successor\_nodes(node.board, node)

# if node is at a maximizing level (if level is even)

if (node.level % 2 == 0):

best\_move = ()

cbv = float("-inf")

# for each successor node, call minimax recursively

for successor in successor\_nodes:

bv\_move = self.minimax(successor)

# look for the highest bv

if bv\_move[0] > cbv:

cbv = bv\_move[0]

best\_move = successor.move

return (cbv, best\_move)

# if node is at a minimizing level (if level is odd)

else:

best\_move = ()

cbv = float("inf")

# for each successor node, call minimax recursively

for successor in successor\_nodes:

bv\_move = self.minimax(successor)

# look for the lowest bv

if bv\_move[0] < cbv:

cbv = bv\_move[0]

best\_move = successor.move

return (cbv, best\_move)

"Minimax with alpha-beta pruning."

def minimax\_alpha\_beta(self, node, A, B):

# if node is at depth limit

if (node.level == node.depth\_limit):

# do a static evaluation, return result and the best move

return (self.static\_eval(node), node.move)

# generate successor nodes

successor\_nodes = self.generate\_successor\_nodes(node.board, node)

# if node is at a maximizing level (if level is even)

if (node.level % 2 == 0):

best\_move = ()

# for each successor node, call minimax recursively

for successor in successor\_nodes:

bv\_move = self.minimax\_alpha\_beta(successor, A, B)

if bv\_move[0] > A:

A = bv\_move[0]

best\_move = successor.move

if A >= B:

return (B, best\_move)

return (A, best\_move)

# if node is at a minimizing level (if level is odd)

else:

best\_move = ()

# for each successor node, call minimax recursively

for successor in successor\_nodes:

bv\_move = self.minimax\_alpha\_beta(successor, A, B)

if bv\_move[0] < B:

B = bv\_move[0]

if B <= A:

return (A, best\_move)

return (B, best\_move)

"""

PLAY THE GAME

"""

def play\_game(width):

"Initialize the board game."

board = BOARD(width)

board.create\_board(width)

winner = None # identifies the winner

user\_piece = None # is User X or O?

computer\_piece = None # is Computer X or O?

user\_turn = None # keeps track of whose turn it is

"Decide who goes first."

who\_first = raw\_input("Who goes first? (Enter 'User' or 'Computer'): ")

if (who\_first == 'User'):

print "User goes first (User is 'X', Computer is 'O')."

user\_piece = 'X'

computer\_piece = 'O'

user\_turn = True

else:

print "Computer goes first (Computer is 'X', User is 'O')."

user\_piece = 'O'

computer\_piece = 'X'

user\_turn = False

"Play the first and second moves."

if (user\_turn):

board.first\_second\_move(user\_turn)

user\_turn = True

else:

board.first\_second\_move(user\_turn)

user\_turn = False

"Start playing!"

# while the game is not over...

while (winner == None):

# if it's the user's turn...

if (user\_turn):

# generate all possible moves for the user

possible\_moves = board.generate\_moves(board.board, user\_piece)

# check if there's a winner (if no moves generates, opponent wins)

if (possible\_moves == []):

winner = 'Computer'

break

# get the move from the user

print "User's turn."

user\_move = board.get\_move(user\_turn, possible\_moves, who\_first)

# make move on the board

board.make\_move(user\_move, user\_piece, True)

user\_turn = False

# if it's the computer's turn...

else:

# generate all possible moves for the user

possible\_moves = board.generate\_moves(board.board, computer\_piece)

# check if there's a winner (if no moves generates, opponent wins)

if (possible\_moves == []):

winner = 'User'

break

# get the move from the computer

print "Computer's turn."

computer\_move = board.get\_move(user\_turn, possible\_moves, who\_first)

# make move on the board

board.make\_move(computer\_move, computer\_piece, True)

user\_turn = True

"Congratulate the winner and end the game."

print winner + " won! Game over."

"Call the play\_game() function."

play\_game(8)