**Pair Programming Equitable Participation & Honesty Affidavit**

We the undersigned promise that we have in good faith attempted to follow the principles of pair programming. Although we were free to discuss ideas with others, the implementation is our own. We have shared a common workspace and taken turns at the keyboard for the majority of the work that we are submitting. Furthermore, any non programming portions of the assignment were done independently. We recognize that should this not be the case, we will be subject to penalties as outlined in the course syllabus.

Pair Programmer 1 (print & sign your name, then date it) 

Pair Programmer 2 (print & sign your name, then date it)

*"""*

*ai - search & strategy module*

*implement a concrete Strategy class and AlphaBetaSearch*

*Name: Tony La and Alexander Giang*

*RedID: 817862169 and 818285579*

*Class Information: CS550: Artificial Intelligence, Spring 2018*

*Professor: Professor Marie Roch*

*Assignment 3: Checkers and AI (Alpha-Beta Pruning)*

*Due Date: 3/13/2018*

*Filename: ai.py*

*"""*

import abstractstrategy

class **Strategy**(abstractstrategy.Strategy):

def **\_\_init\_\_**(*self*, player, game, maxplies):

*self*.player = player

*self*.game = game

*self*.maxplies = maxplies

super(Strategy,*self*).\_\_init\_\_(*self*.player, *self*.game, *self*.maxplies)

*self*.red = 0 #opponent

*self*.black = 1 #AI

def **utility**(*self*, board):

#update pawn and king count

board.update\_counts()

utility = 0

#determine utility based on number of pawns and kings

num\_pawns = board.get\_pawnsN()[*self*.black] - board.get\_pawnsN()[*self*.red]

if num\_pawns >= 0: #black has more

utility += num\_pawns\*3

elif num\_pawns < 0:

utility += num\_pawns\*3

num\_kings = board.get\_kingsN()[*self*.black] - board.get\_kingsN()[*self*.red]

if num\_kings >= 0: #black has more

utility += num\_kings\*5

elif num\_kings < 0:

utility += num\_kings\*5

#Specific locations of where the edges are on the board

edges = ((0,1),(0,3),(0,5),(1,0),(3,0),(5,0),(2,7),(4,7),(6,7),(7,2),(7,4),(7,6))

#specific location of where the corners are

corners = ((0,7), (7,0))

#iterate through the board for information regarding distance to king and locations

for r in range (8):

for c in range (8):

#determine utility based on distance to king

if *self*.maxplayer is board.get(r,c):

if board.disttoking(*self*.player,r) == 1:

utility += 5

elif board.disttoking(*self*.player,r) == 2:

utility += 3

elif board.disttoking(*self*.player,r) == 3:

utility += 1

elif *self*.minplayer is board.get(r,c):

if board.disttoking(*self*.minplayer,r) == 1:

utility -= 5

elif board.disttoking(*self*.minplayer,r) == 2:

utility -= 3

elif board.disttoking(*self*.minplayer,r) == 3:

utility -= 1

#determine utility based on edges and corners

#both will have increase utility but edges will have more

if *self*.maxplayer is board.get(r,c):

if corners.\_\_contains\_\_([r,c]):

utility += 6

elif edges.\_\_contains\_\_([r,c]):

utility += 4

#adjust utility from the consequences of our actions

for actions in board.get\_actions(*self*.maxplayer):

if len(actions[1]) == 3:#indicates we captured a piece

cappiece = board.get(actions[1][2][0], actions[1][2][1])

if board.ispawn(cappiece):

utility += 0.5

elif board.isking(cappiece):

utility += 1

#now look at the case of our next board, whether the piece we move

#or currently have on the board can be captured or not

nextboard = board.move(actions)

for opactions in nextboard.get\_actions(*self*.minplayer):

if len(opactions) >= 3: #indicates multiple capture

utility -=200 #HUGE decrease in utility, make sure to avoid

if len(opactions[1]) == 3: #capture

utility-=20

ourpiece = board.get(opactions[1][2][0], opactions[1][2][1])

if nextboard.ispawn(ourpiece):

utility -= 50

elif nextboard.isking(ourpiece):

utility -= 100

return utility

#does all the searching during playtime

def **play**(*self*, board, hints=True):

board.update\_counts()

if hints:

print(board)

if board.is\_terminal()[0]:

return (board,0)

#instantiate a strategy class

s = Strategy(*self*.player,*self*.game,*self*.maxplies)

#instantiate an alphabetasearch class

albese = AlphaBetaSearch(s,*self*.maxplayer,*self*.minplayer,maxplies=*self*.maxplies)

v = albese.alphabeta(board,*self*.maxplies)

newboard = board.move(v[1])

return (newboard,v[1])

#Class based off of pseudo code in lecture slides

class **AlphaBetaSearch**:

def **\_\_init\_\_**(*self*, strategy, maxplayer, minplayer, maxplies = 3, verbose = False):

*self*.strategy = strategy

*self*.maxplayer = maxplayer

*self*.minplayer = minplayer

*self*.maxplies = maxplies

def **alphabeta**(*self*, state, maxplies):

#finds the b

alpha = float(*'-inf'*)

beta = float(*'inf'*)

v = *self*.max\_value(state,alpha,beta, maxplies)

return v #v = tuple consisting of utility and corresponding action

# finds action with highest utility for AI

def **max\_value**(*self*, state, alpha, beta, plies):

bestaction = []

if state.is\_terminal()[0] or plies <= 0:

v = *self*.strategy.utility(state)

else:

v = float(*'-inf'*)

#compares opponents' (minplayer) utility

for action in state.get\_actions(*self*.maxplayer):

bestaction = action

v = max(v, *self*.min\_value(state.move(action), alpha, beta, plies - 1)[0])

if isinstance(v,int) | isinstance(v,float):

if v >= beta:

break

else:

alpha = max(alpha,v)

else:

if v[0] >= beta:

break

else:

alpha = max(alpha, v[0])

return (v, bestaction)

#finds action with lowest utility for AI

def **min\_value**(*self*, state, alpha, beta, plies):

bestaction = []

if state.is\_terminal()[0] or plies <= 0: #add self.black

v = *self*.strategy.utility(state)

else:

v = float(*'inf'*)

#compares opponents' (minplayer) utility

for action in state.get\_actions(*self*.minplayer):

bestaction = action

v = min(v, *self*.max\_value(state.move(action), alpha, beta, plies - 1)[0])

if isinstance(v,int) | isinstance(v,float):

if v <= alpha:

break

else:

beta = min(beta,v)

else:

if v[0] <= alpha:

break

else:

beta = min(beta, v[0])

return (v, bestaction)

*'''*

*Created on Feb 22, 2015*

**@author:** *mroch*

*Name: Tony La and Alexander Giang*

*RedID: 817862169 and 818285579*

*Class Information: CS550: Artificial Intelligence, Spring 2018*

*Professor: Professor Marie Roch*

*Assignment 3: Checkers and AI (Alpha-Beta Pruning)*

*Due Date: 3/13/2018*

*Filename: checkers.py*

*'''*

import time

import datetime

import ai

import checkerboard

import imp

import sys

major = sys.version\_info[0]

minor = sys.version\_info[1]

modpath = *"\_\_pycache\_\_/tonto.cpython-{}{}.pyc"*.format(major, minor)

tonto = imp.load\_compiled(*"tonto"*, modpath)

# tonto - Professor Roch's not too smart strategy

# You are not given source code to this, but compiled .pyc files

# are available for Python 3.5 and 3.6 (fails otherwise).

# This will let you test some of your game logic without having to worry

# about whether or not your AI is working and let you pit your player

# against another computer player.

#

# Decompilation is cheating, don't do it.

# human - human player, prompts for input

import human

import boardlibrary # might be useful for debugging

def **elapsed**(earlier, later):

*"""elapsed - Convert elapsed time.time objects to duration string*

*Useful for tracking move and game time. Example pseudocode:*

*gamestart = time.time()*

*while game not over:*

*movestart = time.time()*

*... logic ...*

*current = time.time()*

*print("Move time: {} Game time: {}".format(*

*elapsed(movestart, current), elapsed(gamestart, current))*

*"""*

return time.strftime(*'%H:%M:%S'*, time.gmtime(later - earlier))

#in the scenario where we want to play against tonto

#we would set red = tonto.Strategy

def **Game**(red=human.Strategy, black=ai.Strategy, maxplies=8, init=None, verbose=True, firstmove=0):

*"""Game(red, black, maxplies, init, verbose, turn)*

*Start a game of checkers*

*red,black - Strategy classes (not instances)*

*maxplies - # of turns to explore (default 10)*

*init - Start with given board (default None uses a brand new game)*

*verbose - Show messages (default True)*

*firstmove - Player N starts 0 (red) or 1 (black). Default 0.*

*"""*

# Don't forget to create instances of your strategy,

# e.g. black('b', checkerboard.CheckerBoard, maxplies)

maximum\_turns = 40

if init is None:

board = checkerboard.CheckerBoard()

else:

board = init

red\_player = red(*'r'*, board, maxplies)

black\_player = black(*'b'*, board, maxplies)

while not board.is\_terminal()[0] and maximum\_turns > 0:

board.update\_counts()

if firstmove == 0:

#red first

[board, red\_action] = red\_player.play(board)

if verbose:

print(board)

print(*"Red player moved {}"*.format(red\_action))

if red\_action is None: #Player forfeits

print(*"Red forfeited! Black player won!"*)

break

[board, black\_action] = black\_player.play(board)

if verbose:

print(board)

print(*"Black player moved {}"*.format(black\_action))

board.update\_counts()

elif firstmove == 1: #black player first

[board2, black\_action] = black\_player.play(board)

if verbose:

print(board2)

print(*"Black player moved {}"*.format(black\_action))

#red goes

[board1, red\_action] = red\_player.play(board2)

if red\_action is None: #Player forfeits

print(*"Red forfeited! Black player won!"*)

break

if verbose:

print(board1)

print(*"Red player moved {}"*.format(red\_action))

board = board1

maximum\_turns -= 1

if board.is\_terminal()[0]:

#game is over

print (board.is\_terminal()[1],*' won!!'*)

break

if maximum\_turns == 0:

print(*'Game Over! Draw!'*)

break

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

Game()

CS550 Assignment 3

Assignment 3 Part I: Questions (20 points each)

1. Backgammon rolls are between 2 and 12. Find the probability for each pair of numbers. That is without distinguishing between (5,6) and (6,5).
   1. (m,n); m = 1st die from 1 to 6, n = 2nd die from 1 to 6

If m has same probability as n, the pair will be 1/6 \* 1/6 = 1/36 {(1,1), (2,2), (3,3), (4,4), (5,5), (6,6)}

For the rest of the pairs, they will have a probability of 2(1/6 \* 1/6) = 1/18 {all possible roll of dice} – because (1,2) and (2,1) have the same value

* 1. For the same number pairs (i.e (1,1),(2,2), etc), they will have 1/6 \* 1/6 = 1/36 probability. For the rest of them, distinguishing between the pairs don’t matter, so you can half the probabilities 1/36 / 1/2 = 1/18.

1. Explain in your own words the role of chance nodes in stochastic minimax search and how they relate to classic minimax search.
   1. The classic minimax algo is rather somewhat similar to the stochastic minimax algo. The process of getting the favorable value is rather the same. However, with stochastic minimax algo, the utility is weighted with probalistic values. In other words, in the classic minmax algo, it only takes the min or max of the value, while the stochastic way, it takes the min or max of the value consider the combined value with the probability.
   2. Chance nodes in stochastic minimax search have uncertain outcomes, unlike classic minimax search which has a defined max value search and min value search. Because of this uncertainty, the chance node uses the average of the min and max value instead.
2. The CheckerBoard class in the programming assignment below cannot detect stalemates that occur when the board is in the same configuration three different times. How could one modify the class to detect this efficiently? (No implementation is required, just well laid out plan that could be implemented by any skilled computer scientist.)
   1. During a stalemate condition, one would create a list of the past (or history) of the states of the board. When a certain state has been repeated several times (or in this case, reached 3 times) one would end the game and call it a “draw”.
   2. To detect a stalemate condition, a list of past states could be stored. Every time a state is detected, have a counter that adds one. Therefore, the stalemate condition could be detected if the counter reaches three.