"""

Adversarial search algorithms implementation

Your task for homework 5 is to implement:

1. minimax

2. alphabeta

3. abdl (alpha beta depth limited)

"""

**import** random

**import** sys

**def** **rand**(game\_state):

"""

Generate a random move.

:param game\_state: GameState object

:return: a tuple representing the row column of the random move

"""

done = False

**while** **not** done:

row = random.randint(0, game\_state.size - 1)

col = random.randint(0, game\_state.size - 1)

**if** game\_state.available(row,col):

done = True

**return** row, col

**def** **minimax**(game\_state):

"""

Find the best move for our AI agent by applying the minimax algorithm

(searching the entire tree from the current game state)

:param game\_state: GameState object

:return: a tuple representing the row column of the best move

"""

**#Enter your code here**

**def** **value**(game\_state, player):

"""

Calculate the minimax value for any state under the given agent's control

:param game\_state: GameState object - state may be terminal or non-terminal

:param player: (string) 'user' or 'AI' - AI is max

:return: (integer) value of that state -1, 0 or 1

"""

**if** game\_state.is\_win('AI'):

**return** 1

**if** game\_state.is\_win('user'):

**return** -1

**if** game\_state.is\_tie():

**return** 0

# If the agent is MAX return max-value

**if** player **is** 'AI':

**return** max\_value(game\_state)

# If the agent is MIN return min-value

**return** min\_value(game\_state)

**def** **max\_value**(game\_state):

"""

Calculate the minimax value for a non-terminal state under Max's

control (AI agent)

:param game\_state: non-terminal GameState object

:return: (integer) value of that state -1, 0 or 1

"""

v = -sys.maxsize

**for** move **in** game\_state.possible\_moves():

v1 = value(game\_state.successor(move, 'AI'), 'user') # not good

tup = [v, v1]

# print("TUP", tup)

v = max(tup)

# print('MAX: ', v)

**return** v

**def** **min\_value**(game\_state):

"""

Calculate the minimax value for a non-terminal state under Min's

control (user)

:param game\_state: non-terminal GameState object

:return: (integer) value of that state -1, 0 or 1

"""

# Enter your code here and remove the pass statement below

v = sys.maxsize

**for** move **in** game\_state.possible\_moves():

v1 = value(game\_state.successor(move, 'user'), 'AI') # little gud

tup = [v, v1]

# print("TUP", tup)

v = min(tup)

# print('MIN: ', v)

**return** v

**def** **alphabeta**(game\_state):

"""

Find the best move for our AI agent by applying the minimax algorithm

with alpha beta pruning.

:param game\_state: GameState object

:return: a tuple representing the row column of the best move

"""

# Enter your code here and remove the raise statement below\

**#Enter your code here#**

**def** **ab\_value**(game\_state, player, alpha, beta):

"""

Calculate the minimax value for any state under the given agent's control

using alpha beta pruning

:param game\_state: GameState object - state may be terminal or non-terminal

:param player: (string) 'user' or 'AI' - AI is max

:return: (integer) value of that state -1, 0 or 1

"""

# Enter your code here and remove the pass statement below

**if** game\_state.is\_win('AI'):

**return** 1

**if** game\_state.is\_win('user'):

**return** -1

**if** game\_state.is\_tie():

**return** 0

# If the agent is MAX return max-value

**if** player **is** 'AI':

**return** abmax\_value(game\_state, alpha, beta)

# If the agent is MIN return min-value

**return** abmin\_value(game\_state, alpha, beta)

**def** **abmax\_value**(game\_state, alpha, beta):

"""

Calculate the minimax value for a non-terminal state under Max's

control (AI agent) using alpha beta pruning

:param game\_state: non-terminal GameState object

:return: (integer) value of that state -1, 0 or 1

"""

# Enter your code here and remove the pass statement below

a = alpha

v = -sys.maxsize

**for** move **in** game\_state.possible\_moves():

v = max([v, ab\_value(game\_state.successor(move, 'AI'), 'user', a, beta)])

**if** v >= beta:

**return** v

a = max(a, v)

**return** v

**def** **abmin\_value**(game\_state, alpha, beta):

"""

Calculate the minimax value for a non-terminal state under Min's

control (user) using alpha beta pruning

:param game\_state: non-terminal GameState object

:return: (integer) value of that state -1, 0 or 1

"""

# Enter your code here and remove the pass statement below

b = beta

v = sys.maxsize

**for** move **in** game\_state.possible\_moves():

v = min([v, ab\_value(game\_state.successor(move, 'user'), 'AI', alpha, b)])

**if** v <= alpha:

**return** v

b = min([b, v])

**return** v

**def** **abdl**(game\_state, depth):

"""

Find the best move for our AI agent by limiting the alpha beta search to

the given depth and using the evaluation function game\_state.eval()

:param game\_state: GameState object

:return: a tuple representing the row column of the best move

"""

# Enter your code here and remove the raise statement below

**#Enter your code here#**

**def** **abdl\_value**(game\_state, player, alpha, beta, depth):

"""

Calculate the utility for any state under the given agent's control

using depth limited alpha beta pruning and the evaluation

function game\_state.eval()

:param game\_state: GameState object - state may be terminal or non-terminal

:param player: (string) 'user' or 'AI' - AI is max

:return: (integer) utility of that state

"""

# Enter your code here and remove the pass statement below

**if** player == 'AI' **and** game\_state.is\_win('AI'):

**return** 1

**if** player == 'user' **and** game\_state.is\_win('user'):

**return** -1

**if** game\_state.is\_tie():

**return** 0

**if** depth == 0:

**return** game\_state.eval()

# If the agent is MAX return max-value

**if** player **is** 'AI':

**return** abdlmax\_value(game\_state, alpha, beta, depth)

# If the agent is MIN return min-value

**return** abdlmin\_value(game\_state, alpha, beta, depth)

**def** **abdlmax\_value**(game\_state, alpha, beta, depth):

"""

Calculate the utility for a non-terminal state under Max's control

using depth limited alpha beta pruning and the evaluation

function game\_state.eval()

:param game\_state: non-terminal GameState object

:return: (integer) utility (evaluation function) of that state

"""

# Enter your code here and remove the pass statement below

a = alpha

v = -sys.maxsize

**for** move **in** game\_state.possible\_moves():

v = max([v, abdl\_value(game\_state.successor(move, 'AI'), 'user', a, beta, depth -1)])

**if** v >= beta:

**return** v

a = max(a, v)

**return** v

**def** **abdlmin\_value**( game\_state, alpha, beta, depth):

"""

Calculate the utility for a non-terminal state under Min's control

using depth limited alpha beta pruning and the evaluation

function game\_state.eval()

:param game\_state: non-terminal GameState object

:return: (integer) utility (evaluation function) of that state

"""

# Enter your code here and remove the pass statement below

b = beta

v = sys.maxsize

**for** move **in** game\_state.possible\_moves():

v = min([v, abdl\_value(game\_state.successor(move, 'user'), 'AI', alpha, b, depth -1 )])

**if** v <= alpha:

**return** v

b = min([b, v])

**return** v