# Creating a Checkers-Playing Agent Using the Minimax Algorithm with Alpha-Beta Pruning

Jamie Bali

Candidate No: 219057

## Introduction

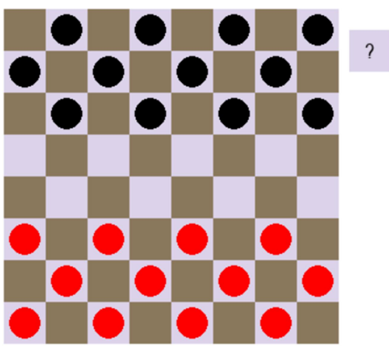
Checkers is a classic game in which two players compete to capture the opponent's pieces, with the victory condition of being the last player standing. We have been tasked with creating an intelligent agent that can play checkers against a human player. Checkers is a game that has been played for centuries, so there are many different rule sets available, but we have been given a specific set of rules to follow. These rules include forced captures, non-forced multi-leg captures, baseline promotion, and the regicide rule. [1]

We decided to write our program using Python. While an alternative language such as Java has better object-orientation and data structures, and may have run slightly faster, we are more comfortable writing programs in Python, and felt that using this language would lead to much cleaner and understandable code.

## Functionality

### Creating the User Interface

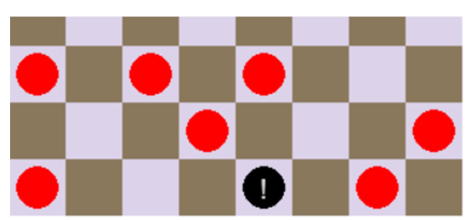
We used pygame as the library for our user interface. Pygame is a free, multi-platform library for Python that allows easy generation of simple user interfaces very quickly. [2]



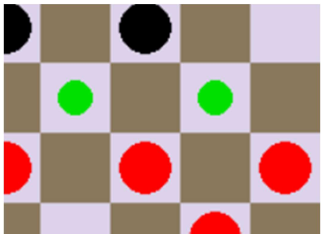

The board is displayed as a series of equally sized tiles of alternating colours, with red and black circles to represent the pieces in the game. Pygame does not create objects for different sprites on the screen, so whenever movement happens on the grid (for example, movement of tiles, displaying valid moves or error text, etc…) the board gets redrawn over the top of the previous one with the changes added (function starts line 401). [Note: Line numbers in this document refer to the line numbers in the appendix of this document, not those in an IDE or on github]

Pygame does not have a button class, so registering what the user is clicking on has to be done by finding the coordinates of the mouse when the user clicks and converting those coordinates into an x and y values to represent which tile they have clicked on. Due to the equal sizes of the tiles, this can be done with a very simple math.floor() call (lines 586-592).

Text is displayed at the bottom of the screen to announce anything that may not be obvious (eg. Promotion, the AI using a multi-leg capture, the user attempting to take an illegal move, etc…). Text is also used to mark which of the pieces on the board are kings. The version of python we were using did not support the ASCII characters we were going to use to show that a tile is a king (black crown♚ and white crown♔) so we had to use an exclamation point to mark that a piece is a king. This works as the pieces stand out, so the user can intuitively figure out that those pieces are kings.



## Allowing the Human to Move



Using the input described in above section, we can highlight the valid moves of a selected tile. Whenever the user clicks on a piece during their turn, a check will happen to see if captures are available, since captures are mandatory if available (lines 684-694). If there is a capture available and the user has clicked on a piece that can't capture another, an error message will be displayed at the bottom of the screen. In any other case, any valid moves from that tile will be displayed (lines 696-721).

When the click happens, a check will happen to see if the user has clicked on has been marked as a valid move. If the tile is marked as a valid move, the piece is moved and any required processing (captures, promotions, deletion of captured pieces) is registered (lines 598-614). By only allowing the player to move onto these highlighted valid move, we prevent the user from moving to an invalid tile. After this, we check for multi-leg captures. We do this by checking if there are captures available from the new location of the moved piece. If so, we gran the user the option to take that capture, but we also give them the capacity to skip the next leg of the capture (lines 617-634). A skip button will appear in the bottom-right corner of the screen, along with a message stating that a multi-leg capture is available if they want to take it After skipping a multi-leg capture or performing a regular move, the agent will get to make a counter move.

## The Intelligent Agent

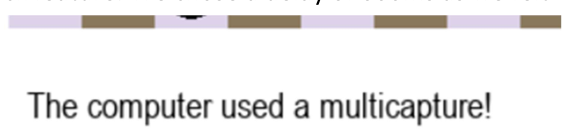
The agent selects a move by performing the minimax algorithm to find the move it can make that will result in the best position a certain number of moves ahead (function starts line 258).

The agent will perform a depth-first search with a depth limit and figure out the value of the board at that point using a set heuristic. The heuristic used figures out the value of a board state based on the positioning of non-king pieces, as well as from the presence of kings. If the agent has a pawn on its own back line, it will increase the value of the board state by 1, but if the agent has a pawn that is one tile away from promoting to a king, then it will be worth 6. Similarly, if the human player has a pawn on their own back line, it will decrease the board value by 1, and if the human player is about to promote to a king, then it will decrease the board value by 6. Having a king on the board will increase or decrease the value of the board by 10 respectively. Since the agent wants to maximise the board value, it will seek to get its pieces closer to promoting to kings and seek to capture any enemy pieces that are close to promoting (lines 240-255).

By picking the maximum value on its moves and the minimum value on the human player's moves, and by thinking up to 8 moves ahead, the AI is able to pick the optimal move it can take with a good degree of accuracy. When a move is passed down the tree so the AI can figure out the next step to take, the whole state of the board is passed down. The board is represented by an 8x8 array of tinyints, taking up a total of 64 bytes. As the tree branches down, the amount of storage used grows massively. We could have represented the board as a 4x8 array, as half of the tiles on the board will never be accessed (all pieces stay on one tile colour), but doing this would have required additional processing that would have decreased the speed of the AI. We thought that it would be better to have a faster AI at the sacrifice of storage, as it totals only a few megabytes of storage at the highest difficulty, which will not cause any issues for a modern computer. Speed, however, will always have a large impact, and the additional couple of seconds that would be added by having to add further if statements to stop the pieces from moving off the awkwardly shaped board would be very impactful on gameplay.

We implemented alpha-beta pruning to increase the speed at which our AI processes moves. Alpha-beta pruning works by removing branches that the agent can prove will not lead to better outcomes. It requires two additional variables (alpha and beta) to be passed down the tree. By comparing these variables to the minimum or maximum at a certain point on the tree, you can confirm that a branch will not provide a better option and is therefore not worth searching further (eg. a branch has already shown to be a worse option for the AI, so there is no point searching it further as you know it won't get picked) (lines 258-359).

The agent also processes multi-leg captures, providing both the single capture, and the multi-leg capture as option to process, thus allowing the AI to skip the move. When the AI returns a multi-leg capture, it sends the final board state, as well as all the legs in-between, back in a list, and they are all displayed one-after-the-other with a delay between. This, along with a message to announce that the agent has used a multi-leg capture, allows the human player to very easily understand what has happened, rather than the AI just jumping half way across the board and multiple pieces disappearing. This delay is added using pygame's clock feature. We chose a delay of 600ms as we felt this was long enough for the human player to figure out what was going on, without causing an unnecessarily long wait.



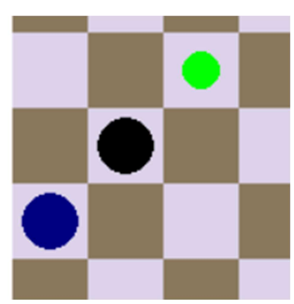

Lastly, we wrote a single line of code that instantly returns a move if there is only 1 move available. If the AI only has one choice of move, there is no point processing the move as it will have to take it no matter what the board values are.

## Difficulty


A title page that appears when the player first loads up the application allows them to select which difficulty they want to play at on a scale from 1 to 6. The difficulty is changed by changing the maximum depth the agent will search. At a difficulty of 1, it will search 3 moves ahead (i.e. the AI's move, the human's move, and then the AI's move again), and at difficulty 6, it'll search 8 moves ahead. These values work well, allowing a scaling difficulty that has a noticeable impact, without being too easy at the easiest difficulty. Having the highest difficulty have a maximum depth of 8 makes it difficult enough without taking too long to process moves. At the highest difficulty, it takes an average of 20 seconds to process the AI's move.

## Hints

We implemented a hint function to help the player out if they are stuck. This was very easy to implement as we already had the minimax function implemented – we just had to run the algorithm from the point of view of the minimising agent (lines 371-388). This then returns a move which we highlight for the player to take if they want. The suggested piece to move is highlighted blue, and the suggested target move is highlighted in green. If the suggested move is a multi-leg move, we only highlight the first leg, as this will be easier for the human player to understand than just highlighting where it would end after the multiple steps.



# Appendix

```
import pygame
import math
import numpy as np

pygame.init()
screen = pygame.display.set_mode((400,400))
clock = pygame.time.Clock()
font = pygame.font.SysFont("Arial", 20)
fontB = pygame.font.SysFont("Arial", 20, bold=True)

###
#
# When generating the possible moves, we will first look for captures as they must take precedence.
# Since captures are manditory, if the possible moves stack is empty after this processing, we will
# perform a second run to get regular moves. This is slightly less efficient time-wise, but uses less
# storage space.
#
# For ease of processing, we will return the board state after each move, as opposed to the move itself.
#
###
def getPossibleMoves(inboard, player, mc = "False", mcfiller = [0]):
    moves = []
    tempBoard = np.copy(inboard)

    while len(tempBoard) != 8:
        tempBoard = tempBoard[0]

    board = tempBoard
    if player == 1: # player 1 is the human player.
        captures = False
        for i in range(0,8):          # player 1 captures
            for j in range(0,8):
                if board[i][j] == 4 or board[i][j] == 1: # king or regular
                    if i > 1 and j < 6:              # to avoid overflow errors
                        if (board[i-1][j+1] == 2 or board[i-1][j+1] == 5) and board[i-2][j+2] == 0: # if next to an enemy tile, and beyond that is empty, we know we can take that piece
                            newBoardState = np.copy(board)
                            if newBoardState[i-1][j+1] == 5:       # this implements the regicide rule.
                                newBoardState[i-2][j+2] == 4
                            else:
                                newBoardState[i-2][j+2] = board[i][j]   # update movement
                            newBoardState[i][j] = 0        # clear previous points
                            newBoardState[i-1][j+1] = 0
                            if i-2 == 0:                    # this checks for promotion
                                newBoardState[i-2][j+2] = 4
                            captures = True                 # this confirms that a piece was taken, so we know that non-captures aren't allowed
                            moves.append([newBoardState, mcfiller]) # mcfiller means there won't be errors in the multicapture system.
                            mcmoves = getPossibleMoves(newBoardState, 1, "True", newBoardState)     # recursion for multicaptures, as the AI doesn't always want to multicapture
                            for x in mcmoves:                   # all multicapture options are added as seperate moves.
                                moves.append([x,mcfiller])
                    if i > 1 and j > 1:                                      # this repeats as above for all other capture directions
                        if (board[i-1][j-1] == 2 or board[i-1][j-1] == 5) and board[i-2][j-2] == 0:
```

```
                    newBoardState = np.copy(board)
                    if newBoardState[i-1][j-1] == 5:
                        newBoardState[i-2][j-2] == 4
                    else:
                        newBoardState[i-2][j-2] = board[i][j]
                    newBoardState[i][j] = 0
                    newBoardState[i-1][j-1] = 0
                    if i-2 == 0:
                        newBoardState[i-2][j-2] = 4
                    captures = True
                    moves.append([newBoardState, mcfiller])
                    mcmoves = getPossibleMoves(newBoardState, 1, "True", newBoardState)
                    for x in mcmoves:
                        moves.append([x,mcfiller])
        if board[i][j] == 4: # backwards moves means king only.
            if i < 6 and j < 6:
                if (board[i+1][j+1] == 2 or board[i+1][j+1] == 5) and board[i+2][j+2] == 0:
                    newBoardState = np.copy(board)
                    if newBoardState[i+1][j+1] == 5:
                        newBoardState[i+2][j+2] == 4
                    else:
                        newBoardState[i+2][j+2] = board[i][j]
                    newBoardState[i][j] = 0
                    newBoardState[i+1][j+1] = 0
                    captures = True
                    moves.append([newBoardState, mcfiller])
                    mcmoves = getPossibleMoves(newBoardState, 1, "True", newBoardState)
                    for x in mcmoves:
                        moves.append([x,mcfiller])
            if i < 6 and j > 1:
                if (board[i+1][j-1] == 2 or board[i+1][j-1] == 5) and board[i+2][j-2] == 0:
                    newBoardState = np.copy(board)
                    if newBoardState[i+1][j-1] == 5:
                        newBoardState[i+2][j-2] == 4
                    else:
                        newBoardState[i+2][j-2] = board[i][j]
                    newBoardState[i][j] = 0
                    newBoardState[i+1][j-1] = 0
                    captures = True
                    moves.append([newBoardState, mcfiller])
                    mcmoves = getPossibleMoves(newBoardState, 1, "True", newBoardState)
                    for x in mcmoves:
                        moves.append([x, mcfiller])

if captures == False and mc == "False":           # player 1 non captures
    for i in range(0,8):
        for j in range(0,8):
            if board[i][j] == 4 or board[i][j] == 1: # forward moves for both king and normal.
                if i > 0 and j < 7:
                    if board[i-1][j+1] == 0:
                        newBoardState = np.copy(board)
                        newBoardState[i-1][j+1] = newBoardState[i][j]
                        newBoardState[i][j] = 0
                        moves.append([newBoardState, mcfiller]) # the filler is still addded so less processing is needed later.
                if i > 0 and j > 0:
```

```
                            if board[i-1][j-1] == 0:
                                newBoardState = np.copy(board)
                                newBoardState[i-1][j-1] = newBoardState[i][j]
                                newBoardState[i][j] = 0
                                moves.append([newBoardState, mcfiller])
                    if board[i][j] == 4: # king only
                        if i < 7 and j < 7:
                            if board[i+1][j+1] == 0:
                                newBoardState = np.copy(board)
                                newBoardState[i+1][j+1] = newBoardState[i][j]
                                newBoardState[i][j] = 0
                                moves.append([newBoardState, mcfiller])
                        if i < 7 and j > 0:
                            if board[i+1][j-1] == 0:
                                newBoardState = np.copy(board)
                                newBoardState[i+1][j-1] = newBoardState[i][j]
                                newBoardState[i][j] = 0
                                moves.append([newBoardState, mcfiller])

    else: # player 2 is the artifical agent. Uses the same processing as above, so no comments have been written.
        captures = False

        for i in range(0,8):              # AI Captures
            for j in range(0,8):
                if board[i][j] == 2 or board[i][j] == 5: # king or regular
                    if i < 6 and j < 6:
                        if (board[i+1][j+1] == 1 or board[i+1][j+1] == 4) and board[i+2][j+2] == 0: # 1 is a human piece and 5 is a human king.
                            newBoardState = np.copy(board)
                            if board[i+1][j+1] == 4:
                                newBoardState[i+2][j+2] = 5
                            else:
                                newBoardState[i+2][j+2] = newBoardState[i][j]
                            newBoardState[i][j] = 0
                            newBoardState[i+1][j+1] = 0
                            if i+2 == 7:
                                newBoardState[i+2][j+2] = 5
                            captures = True
                            moves.append([newBoardState, mcfiller])
                            mcmoves = getPossibleMoves(newBoardState, 2, "True", newBoardState)
                            for x in mcmoves:
                                moves.append([x,mcfiller])
                    if i < 6 and j > 1:
                        if (board[i+1][j-1] == 1 or board[i+1][j-1] == 4) and board[i+2][j-2] == 0:
                            newBoardState = np.copy(board)
                            if board[i+1][j-1] == 4:
                                newBoardState[i+2][j-2] = 5
                            else:
                                newBoardState[i+2][j-2] = newBoardState[i][j]
                            newBoardState[i][j] = 0
                            newBoardState[i+1][j-1] = 0
                            if i+2 == 7:
                                newBoardState[i+2][j-2] = 5
                            captures = True
                            moves.append([newBoardState, mcfiller])
                            mcmoves = getPossibleMoves(newBoardState, 2, "True", newBoardState)
```

```
                    for x in mcmoves:
                        moves.append([x,mcfiller])
        if board[i][j] == 5: # king only
            if i > 1 and j > 1:
                if (board[i-1][j-1] == 1 or board[i-1][j-1] == 4) and board[i-2][j-2] == 0:
                    newBoardState = np.copy(board)
                    newBoardState[i-2][j-2] = newBoardState[i][j]
                    newBoardState[i][j] = 0
                    newBoardState[i-1][j-1] = 0
                    captures = True
                    moves.append([newBoardState, mcfiller])
                    mcmoves = getPossibleMoves(newBoardState, 2, "True", newBoardState)
                    for x in mcmoves:
                        moves.append([x,mcfiller])
            if i > 1 and j < 6:
                if (board[i-1][j+1] == 1 or board[i-1][j+1] == 4) and board[i-2][j+2] == 0:
                    newBoardState = np.copy(board)
                    newBoardState[i-2][j+2] = newBoardState[i][j]
                    newBoardState[i][j] = 0
                    newBoardState[i-1][j+1] = 0
                    captures = True
                    moves.append([newBoardState, mcfiller])
                    mcmoves = getPossibleMoves(newBoardState, 2, "True", newBoardState)
                    for x in mcmoves:
                        moves.append([x,mcfiller])

if captures == False and mc == "False":             # AI non captures
    for i in range (0,8):
        for j in range(0,8):
            if board[i][j] == 2 or board[i][j] == 5: # king or regular
                if i < 7 and j < 7:
                    if board[i+1][j+1] == 0:
                        newBoardState = np.copy(board)
                        newBoardState[i+1][j+1] = board[i][j]
                        newBoardState[i][j] = 0
                        if i+1 == 7:
                            newBoardState[i+1][j+1] = 5
                        moves.append([newBoardState, mcfiller])

                if i < 7 and j > 0:
                    if board[i+1][j-1] == 0:
                        newBoardState = np.copy(board)
                        newBoardState[i+1][j-1] = board[i][j]
                        newBoardState[i][j] = 0
                        if i+1 == 7:
                            newBoardState[i+1][j-1] = 5
                        moves.append([newBoardState, mcfiller])
            if board[i][j] == 5: # king only
                if i > 0 and j < 7:
                    if board[i-1][j+1] == 0:
                        newBoardState = np.copy(board)
                        newBoardState[i-1][j+1] = board[i][j]
                        newBoardState[i][j] = 0
                        moves.append([newBoardState, mcfiller])
                if i > 0 and j > 0:
```

```
                            if board[i-1][j-1] == 0:
                                newBoardState = np.copy(board)
                                newBoardState[i-1][j-1] = board[i][j]
                                newBoardState[i][j] = 0
                                moves.append([newBoardState, mcfiller])
    return moves

###
#
# The game playing agent. When initialised, it'll need to take the difficulty.
# Each turn, the board state will be passed to the agent and it use the minimax algorithm
# with alpha-beta pruning to find an optimal move.
# The moves will be returned in the form of a tuple (or maybe a list, whichever is easier to implement),
# with the structure (from, to, ...) where the tuple will increase in size of any multiple captures the AI
# performs.
#
###
class Agent:
    def __init__(self, difficulty = 2):
        self.maxDepth = difficulty          # the difficulty refers to how deep the AI will search.

    def getBoardValue(self, board):
        value = 0
        while len(board) != 8:  # only gets the most up to date version of the board
                                # ignoring the multicapture inbetweens.
            board = board[0]
        for i in range (0,8):           # for each tile on the board
            for j in range (0,8):
                if board[i][j] == 1:
                    value -= (7-i)      # the regular pieces are worth more if they are closer to becoming kings
                elif board[i][j] == 2:
                    value += i          # same as above, but for the AI's pieces
                elif board[i][j] == 4:
                    value -= 10         # 4 is an enemy king. The position is irrelevant, only that it is a king
                elif board[i][j] == 5:
                    value += 10         # 5 is an ally king, meaning we want as many of these as possible.
        return value


    def minimax(self, boardState, player, depth, maxDepth, alpha, beta):
        self.alpha = alpha
        self.beta = beta
        self.maxDepth = maxDepth
        self.depth = depth
        self.boardState = boardState
        self.player = player

        self.moves = getPossibleMoves(self.boardState, self.player)    # start by getting all the valid moves it could take at a point.

        self.minval = 100
        self.maxval = -100

        self.breaker = False
        self.x = 0

```

```
        ### BOTTOM LAYER OF TREE
        if self.depth == 1:
            while self.x < len(self.moves):                    # the breaker exists so we don't have to use a break command to
                                                               # exit the loop when the path is pruned.
                self.temp = self.getBoardValue(self.moves[self.x][0])       # get value of a board state after a certain move
                if self.player == 2:            # meaning it's the max agent
                    if self.temp > self.maxval:
                        self.maxval = self.temp
                    if self.temp > self.alpha:
                        self.alpha = self.temp
                    if self.alpha >= self.beta:      # this is the alpha-beta pruning check
                        self.breaker = True
                if self.player == 1:            # meaning it's the min agent
                    if self.temp < self.minval:
                        self.minval = self.temp
                    if self.temp < self.beta:
                        self.beta = self.temp
                    if self.alpha >= self.beta:
                        self.breaker = True
                self.x += 1

            if self.player == 2:
                return self.maxval, self.alpha      # the max agent wants to return the alpha
            else:
                return self.minval, self.beta       # and the min agent wants to return the beta

        ### MIDDLE LAYERS OF TREE
        elif self.depth != self.maxDepth:
            self.agent = Agent()
            while self.x < len(self.moves) and self.breaker == False:

                # this if statement makes sure the pruning functions correctly, by modifying alpha and beta respective to which agent is processing.
                if player == 1:
                    self.temp, self.beta = self.agent.minimax(self.moves[self.x][0], (self.player%2)+1, self.depth - 1, self.maxDepth, self.alpha, self.beta)
                else:
                    self.temp, self.alpha = self.agent.minimax(self.moves[self.x][0], (self.player%2)+1, self.depth - 1, self.maxDepth, self.alpha,
self.beta)


                if self.player == 2:            # Max agent
                    if self.temp > self.maxval:
                        self.maxval = self.temp
                    if self.temp > self.alpha:
                        self.alpha = self.temp
                    if self.alpha >= self.beta:
                        self.breaker = True
                if self.player == 1:            # Min agent
                    if self.temp < self.minval:
                        self.minval = self.temp
                    if self.temp < self.beta:
                        self.beta = self.temp
                    if self.alpha >= self.beta:
                        self.breaker = True
                self.x += 1
```

```
            if self.player == 2:
                return self.maxval, self.alpha
            else:
                return self.minval, self.beta

        ### ROOT OF TREE
        else:
            if len(self.moves) == 0:    # if there are no moves available, then the AI has lost.
                return "Loss"
            if len(self.moves) == 1:    # if there is only 1 move available, then we don't need to run the minimax algorithm
                return self.moves[0]

            self.agent = Agent()
            self.bestIndex = -1

            while self.x < len(self.moves):    # there is no breaker here, as alpha-beta pruning doesn't function on the root node
                if player == 2:
                    self.temp, self.alpha = self.agent.minimax(self.moves[self.x][0], (self.player%2)+1, self.depth - 1, self.maxDepth, self.alpha, self.beta)
                else:
                    self.temp, self.beta = self.agent.minimax(self.moves[self.x][0], (self.player%2)+1, self.depth - 1, self.maxDepth, self.alpha, self.beta)
                if player == 2:
                    if self.temp > self.maxval:
                        self.maxval = self.temp
                        self.bestIndex = self.x     # keeps an index of the best move.
                else:
                    if self.temp < self.minval:
                        self.minval = self.temp
                        self.bestIndex = self.x
                self.x += 1
            return self.moves[self.bestIndex] # returns the move with the best value

    ###
    #
    # The move function will take the board state and run minimax on it to generate an optimal move.
    #
    ###
    def move(self, boardState):
        self.boardState = boardState
        stateOfChosen = self.minimax(self.boardState, 2, self.maxDepth, self.maxDepth, -100, 100)
        return stateOfChosen

    def hint(self, boardState):
        self.boardState = boardState
        self.stateOfChosen = self.minimax(self.boardState, 1, self.maxDepth, self.maxDepth, -100, 100)[0]

        if len(self.stateOfChosen) != 8:
            self.stateOfChosen = self.stateOfChosen[len(self.stateOfChosen)-1]
            print(self.stateOfChosen)

        hx = 0
        hy = 0

        for x in range(0,8):
                for y in range(0,8):
```

```
                    if (self.stateOfChosen[x][y] == 1 or self.stateOfChosen[x][y] == 4) and self.boardState[x][y] == 0:
                        self.boardState[x][y] = 9   # we are using 9 to mark the suggested move.
                    if self.stateOfChosen[x][y] == 0 and (self.boardState[x][y] == 1 or self.boardState[x][y] == 4):
                        hx = x
                        hy = y



        return self.boardState, hx, hy

def clearBoard(board):
    for i in range(0,8): # clear board
        for j in range(0,8):
            if board[i][j] == 3 or board[i][j] == 9:    # 3 is the valid moves that get highlighted
                                                        # 9 is the suggested hint
                board[i][j] = 0

def drawBoard(board, hx = -1, hy = -1):

    screen.fill((255,255,255))      # fill screen in white. This also covers the previous drawings so they can be redisplayed correctly

    darkSquare = (138,120,93)
    lightSquare = (220,211,234)

    for x in range(0,8):
        for y in range(0,8):
            if x % 2 == 1:
                if y % 2 == 1:  # the mod operator means that each alternating tile is highted a different colour.
                    pygame.draw.rect(screen, darkSquare, pygame.Rect(10 + (40*x),10 + (40*y),40,40))
                else:
                    pygame.draw.rect(screen, lightSquare, pygame.Rect(10 + (40*x),10 + (40*y),40,40))
            else:
                if y % 2 == 1:  # as above.
                    pygame.draw.rect(screen, lightSquare, pygame.Rect(10 + (40*x),10 + (40*y),40,40))
                else:
                    pygame.draw.rect(screen, darkSquare, pygame.Rect(10 + (40*x),10 + (40*y),40,40))

    # this rectangle is drawn as the hint button.
    pygame.draw.rect(screen, lightSquare, pygame.Rect(340, 30, 40, 40))
    txt = font.render("?", 1, (0,0,0))
    screen.blit(txt, (355, 38))

    for x in range(0,8):
        for y in range(0,8):
            if board[y][x] == 1:
                pygame.draw.circle(screen, (255,0,0), ((x*40)+30,(y*40)+30),15) # red circle for human regular piece
            elif board[y][x] == 2:
                pygame.draw.circle(screen, (0,0,0), ((x*40)+30,(y*40)+30),15)   # black circle for AI regular piece
            elif board[y][x] == 3:
                pygame.draw.circle(screen, (0,225,0), ((x*40)+30,(y*40)+30),10) # smaller green circle for valid move highlights
            elif board[y][x] == 4:
                pygame.draw.circle(screen, (255,0,0), ((x*40)+30,(y*40)+30),15) # red circle with ! for human king
                king = fontB.render("!", 1, (0,0,0))
                screen.blit(king, ((x*40)+27,(y*40)+19))
            elif board[y][x] == 5:
```

```
                pygame.draw.circle(screen, (0,0,0), ((x*40)+30,(y*40)+30),15)   # black circle with ! for AI king
                king = fontB.render("!", 1, (225,225,225))
                screen.blit(king, ((x*40)+27,(y*40)+19))
            elif board[y][x] == 9:
                pygame.draw.circle(screen, (0,255,0), ((x*40)+30,(y*40)+30),10) # smaller green circle for hint

    if hx > -1 and hy > -1:
        pygame.draw.circle(screen, (0,0,128), ((hx*40)+30,(hy*40)+30),15) # blue circle for hint start point

def capturesAvailable(board):
    captures = False
    for a in range(0,8):           # go through all possible moves to see if a valid capture is available
        for b in range(0,8):
            if (board[a][b] == 1 or board[a][b] == 4) and a > 1 and b < 6:      # if tile contains human piece, and capture wouldn't cause overflow
                if board[a-1][b+1] == 2 or board[a-1][b+1] == 5:              # if diagonal tile contains AI piece
                    if board[a-2][b+2] == 0:                                  # and if tile beyond there is empty
                        captures = True                                       # then a capture is available
            if (board[a][b] == 1 or board[a][b] == 4) and a > 1 and b > 1:      # repeat for all possible moves the human has
                if board[a-1][b-1] == 2 or board[a-1][b-1] == 5:
                    if board[a-2][b-2] == 0:
                            captures = True
            if board[a][b] == 4 and a < 6 and b < 6:
                if board[a+1][b+1] == 2 or board[a+1][b+1] == 5:
                    if board[a+2][b+2] == 0:
                        captures = True
            if board[a][b] == 4 and a < 6 and b > 1:
                if board[a+1][b-1] == 2 or board[a+1][b-1] == 5:
                    if board[a+2][b-2] == 0:
                        captures = True
    return captures

def drawTitlePage(diff):
    darkSquare = (138,120,93)
    lightSquare = (220,211,234)

    screen.fill((255,255,255))
    titleText = font.render("Checkers !", 1, (0,0,0))
    screen.blit(titleText, (150,80))
    diffText = font.render("Select your difficulty!", 1, (0,0,0))
    screen.blit(diffText, (110,150))

    # these here are the difficulty buttons
    pygame.draw.rect(screen, lightSquare, pygame.Rect(50,200,40,40))
    pygame.draw.rect(screen, lightSquare, pygame.Rect(102,200,40,40))
    pygame.draw.rect(screen, lightSquare, pygame.Rect(154,200,40,40))
    pygame.draw.rect(screen, lightSquare, pygame.Rect(206,200,40,40))
    pygame.draw.rect(screen, lightSquare, pygame.Rect(258,200,40,40))
    pygame.draw.rect(screen, lightSquare, pygame.Rect(310,200,40,40))

    # this here highlights the selected difficulty in a darker colour
    if diff == 1:
        pygame.draw.rect(screen, darkSquare, pygame.Rect(50,200,40,40))
    elif diff == 2:
        pygame.draw.rect(screen, darkSquare, pygame.Rect(102,200,40,40))
    elif diff == 3:
```

```
            pygame.draw.rect(screen, darkSquare, pygame.Rect(154,200,40,40))
        elif diff == 4:
            pygame.draw.rect(screen, darkSquare, pygame.Rect(206,200,40,40))
        elif diff == 5:
            pygame.draw.rect(screen, darkSquare, pygame.Rect(258,200,40,40))
        else:
            pygame.draw.rect(screen, darkSquare, pygame.Rect(310,200,40,40))

        # this puts the numbers onto the difficulty buttons
        numText = font.render("1", 1, (0,0,0))
        screen.blit(numText, (66,208))
        numText = font.render("2", 1, (0,0,0))
        screen.blit(numText, (118,208))
        numText = font.render("3", 1, (0,0,0))
        screen.blit(numText, (170,208))
        numText = font.render("4", 1, (0,0,0))
        screen.blit(numText, (222,208))
        numText = font.render("5", 1, (0,0,0))
        screen.blit(numText, (274,208))
        numText = font.render("6", 1, (0,0,0))
        screen.blit(numText, (326,208))

        # this creates the "play game" button
        pygame.draw.rect(screen, lightSquare, pygame.Rect(100, 300, 200, 50))
        goText = font.render("Let's Play!", 1, (0,0,0))
        screen.blit(goText, (160,312))

###
#
# Main function. Takes no inputs.
# This is where the checkers game will be run from.
#
###
if __name__ == '__main__':
    title = True
    difficulty = 3
    while title:
        drawTitlePage(difficulty)
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                exit()
            elif event.type == pygame.MOUSEBUTTONDOWN:
                if pygame.mouse.get_pressed()[0]:                    # if player right clicks on a button, difficulty changes
                    x, y = pygame.mouse.get_pos()
                    if x > 100 and x < 300 and y > 300 and y < 350:  # these coordinates are for the "play game" button
                        title = False
                    elif y > 200 and y < 240:                    # this coordinates are for the respective buttons.
                        if x > 50 and x < 90:
                            difficulty = 1
                        elif x > 102 and x < 142:
                            difficulty = 2
                        elif x > 154 and x < 194:
                            difficulty = 3
                        elif x > 206 and x < 246:
```

```
                        difficulty = 4
                    elif x > 258 and x < 298:
                        difficulty = 5
                    elif x > 310 and x < 350:
                        difficulty = 6
        clock.tick(30)
        pygame.display.update()


# the difficulty is the selected value +2, as thinking only 1 move ahead would be too easy at the start, and we want difficulty to scale linearly
difficulty = difficulty + 2
agent = Agent(difficulty)

pastClick = (-1,-1)

# this block creates the initial board state
board = []
board.append([0,2,0,2,0,2,0,2])
board.append([2,0,2,0,2,0,2,0])
board.append([0,2,0,2,0,2,0,2])
board.append([0,0,0,0,0,0,0,0])
board.append([0,0,0,0,0,0,0,0])
board.append([1,0,1,0,1,0,1,0])
board.append([0,1,0,1,0,1,0,1])
board.append([1,0,1,0,1,0,1,0])
drawBoard(board)

mcavailable = False

gameRunning = 0

while gameRunning == 0:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            exit()
        elif event.type == pygame.MOUSEBUTTONDOWN:
            if pygame.mouse.get_pressed()[0]: # confirm that it is a left click.
                dy, dx = pygame.mouse.get_pos()

                # get which tile was clicked
                x = math.floor((dx-10)/40)
                y = math.floor((dy-10)/40)


                if x >= 0 and x < 8 and y >= 0 and y < 8:   # as long as it is a valid tile
                    wascap = False
                    moved = False
                    if board[x][y] == 3:                 # check if it was a valid movement
                        board[x][y] = board[pastClick[0]][pastClick[1]]     # if so, make the move
                        board[pastClick[0]][pastClick[1]] = 0
                        if x == 0:                                 # check for promotion
                            board[x][y] = 4
                            Text = font.render("PROMOTION!", 1, (0,0,0))
```

```
    screen.blit(Text, (20,360))
clearBoard(board)
temp = x-pastClick[0]
if abs(temp) == 2: # this means it was a capture
    wascap = True                # wasCapture is used for multicapture capability
    dx = int((x - pastClick[0])/2)
    dy = int((y - pastClick[1])/2)
    if board[x-dx][y-dy] == 5: # this line implements regicide
        board[x][y] = 4
    board[x-dx][y-dy] = 0
moved = True

mcavailable = False
if wascap:          # if it was a capture, we check if multicapture is possible.
    if (board[x][y] == 4 or board[x][y] == 1) and x > 1:
        if y > 1:
            if (board[x-1][y-1] == 2 or board[x-1][y-1] == 5) and board[x-2][y-2] == 0:
                mcavailable = True
        if y < 6:
            if (board[x-1][y+1] == 2 or board[x-1][y+1] == 5) and board[x-2][y+2] == 0:
                mcavailable = True
    if board[x][y] == 4 and x < 6:
        if y > 1:
            if (board[x+1][y-1] == 2 or board[x+1][y-1] == 5) and board[x+2][y-2] == 0:
                mcavailable = True
        if y < 6:
            if (board[x+1][y+1] == 2 or board[x+1][y+1] == 5) and board[x+2][y+2] == 0:
                mcavailable = True
if not mcavailable: # if it wasn't a multicapture, we go straight to running the AI
    pastClick = (-1,-1)
    clearBoard(board)
    drawBoard(board)

    Text = font.render("I'm thinking...", 1, (0,0,0))
    screen.blit(Text, (20,360))

    pygame.display.update()

    # AI
    agentMove = agent.move(board)
    if agentMove == "Loss":
        # the human has won
        gameRunning = 1
    else:
        agentMove = agentMove[0]

    if len(agentMove) != 8:              # this is how the AI does multicaptures
        for mcmoves in range(1, len(agentMove)+1):       # we itterate throught he AI's multicapture
                                                          # steps and display them all seperately
            board = agentMove[len(agentMove) - mcmoves]
            drawBoard(board)
            pygame.display.update()
            pygame.time.delay(600)          # we found that 600 ms is about long enough of a delay between steps
        errorText = font.render("The computer used a multicapture!", 1, (0,0,0))    # announce what happened
        screen.blit(errorText, (20,360))
```

```
        else:                         # if the AI doens't multicapture, we just display the move
            board = agentMove
            drawBoard(board)
    else:
        clearBoard(board)         # redisplay the board without the green markers if no move was made
        drawBoard(board)
        pygame.display.update()

else:
    pastClick = (x,y)

clearBoard(board)

if mcavailable:
    errorText = font.render("There is a valid multicapture available!", 1, (0,0,0))
    screen.blit(errorText, (20,360))
    pygame.draw.rect(screen, (220,211,234), (pygame.Rect(340, 340, 40, 40)))   # show the skip button if the user doesn't want to multicapture
    txt = font.render("Skip", 1, (0,0,0))
    screen.blit(txt, (341,344))
    pygame.display.update()

if moved == False: # mark valid moves
    captures = capturesAvailable(board)

    if captures == False:         # if there wasn't a capture, then any valid movement is a valid move
        if board[x][y] == 1 or board[x][y] == 4:
            if x > 0 and y < 7 and board[x-1][y+1] == 0:
                board[x-1][y+1] = 3
            if x > 0 and y > 0 and board[x-1][y-1] == 0:
                board[x-1][y-1] = 3
        if board[x][y] == 4:
            if x < 7 and y < 7 and board[x+1][y+1] == 0:
                board[x+1][y+1] = 3
            if x < 7 and y > 0 and board[x+1][y-1] == 0:
                board[x+1][y-1] = 3

    if captures == True:          # if there was a capture, only captures are valid moves
        valid = False   # this is a tracker to see if you highlighted a valid move so i can provide an error message
        if board[x][y] == 1 or board[x][y] == 4:
            if x > 1 and y < 6 and (board[x-1][y+1] == 2 or board[x-1][y+1] == 5) and board[x-2][y+2] == 0:
                board[x-2][y+2] = 3
                valid = True
            if x > 1 and y > 1 and (board[x-1][y-1] == 2 or board[x-1][y-1] == 5) and board[x-2][y-2] == 0:
                board[x-2][y-2] = 3
                valid = True
        if board[x][y] == 4:
            if x < 6 and y < 6 and (board[x+1][y+1] == 2 or board[x+1][y+1] == 5) and board[x+2][y+2] == 0:
                board[x+2][y+2] = 3
                valid = True
            if x < 6 and y > 1 and (board[x+1][y-1] == 2 or board[x+1][y-1] == 5) and board[x+2][y-2] == 0:
                board[x+2][y-2] = 3
                valid = True
```

```
        drawBoard(board) # update the green movement tiles
        if mcavailable:
            errorText = font.render("There is a valid multicapture available!", 1, (0,0,0)) # show an error maessage
            screen.blit(errorText, (20,360))
            pygame.draw.rect(screen, (220,211,234), (pygame.Rect(340, 340, 40, 40)))
            txt = font.render("Skip", 1, (0,0,0))
            screen.blit(txt, (341,344))
            pygame.display.update()

        if captures == True and valid == False and mcavailable == False:
            errorText = font.render("There is a valid capture available!", 1, (0,0,0))  # show an error message
            screen.blit(errorText, (20,360))

    captures = False
elif dy > 340 and dy < 380 and dx > 30 and dx < 70:     # these are the coordinates of the hint button
    clearBoard(board)
    drawBoard(board)
    txt = font.render("Let's have a look for you!", 1, (0,0,0))     # anounce that it is searching
    screen.blit(txt, (20,360))
    pygame.display.update()
    board, hx, hy = agent.hint(board)
    drawBoard(board, hy, hx)
    txt = font.render("Try moving here!", 1, (0,0,0))             # this hint will just show where the best move would end up
    screen.blit(txt, (20,360))

elif dy > 340 and dy < 380 and dx > 340 and dx < 380 and mcavailable == True:   # this is the skip button, but only if it is showing
    mcavailable = False
    pastClick = (-1,-1)
    clearBoard(board)
    drawBoard(board)
    Text = font.render("I'm thinking...", 1, (0,0,0))
    screen.blit(Text, (20,360))
    pygame.display.update()
    # AI
    agentMove = agent.move(board)
    if agentMove == "Loss":
        # the human has won
        gameRunning = 1
    else:
        agentMove = agentMove[0]

    if len(agentMove) != 8:
        for mcmoves in range(1, len(agentMove)+1):
            board = agentMove[len(agentMove) - mcmoves]
            drawBoard(board)
            pygame.display.update()
            pygame.time.delay(600)
        errorText = font.render("The computer used a multicapture!", 1, (0,0,0))
        screen.blit(errorText, (20,360))
    else:
        board = agentMove
        drawBoard(board)
x = -1
y = -1
```

```
    loss = True
    for x in range(0,8):                        # this is to check if the player has lost yet.
        for y in range(0,8):
            if board[x][y] == 1 or board[x][y] == 4:    # no human pieces on the board means they have lost.
                loss = False
    if loss:
        gameRunning = 2
    clock.tick(30)
    pygame.display.update()


ender = False

while ender == False:
    if gameRunning == 1:
        errorText = font.render("CONGRATULATIONS! YOU WON!", 1, (0,0,0))    # announce victory or loss
        screen.blit(errorText, (20,360))
    else:
        errorText = font.render("THE COMPUTER WINS!", 1, (0,0,0))
        screen.blit(errorText, (20,360))
    pygame.display.update()                         # clicking anywhere ends the game and shuts the program
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            ender = True
        elif event.type == pygame.MOUSEBUTTONDOWN:
            pygame.quit()
            ender = True

### END
```

Copies of the code, readme file, and this report can be found at:
https://github.com/JamieBali/checkersMinimax

## References

[1] https://a4games.company/checkers-rules-and-layout/

[2] https://www.pygame.org/docs/