# Implementation of Checkers with a Minimax Function

A report

*by*

Jesper de Jonge

198753

Dr. Ronald Grau

December 2020

UNIVERSITY OF SUSSEX

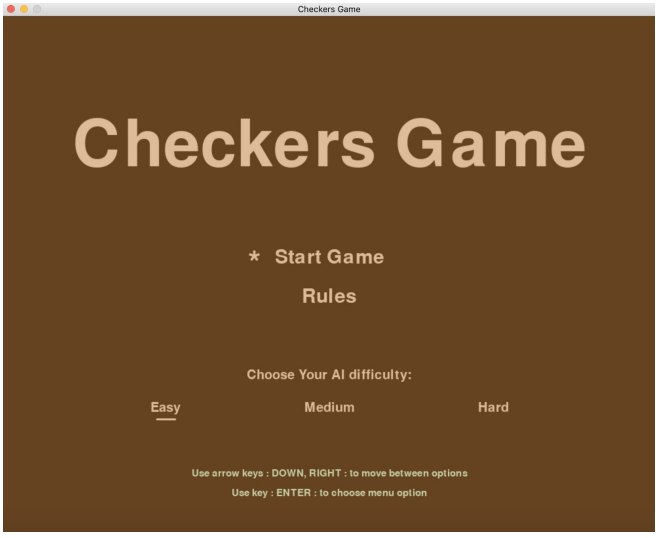
# Introduction

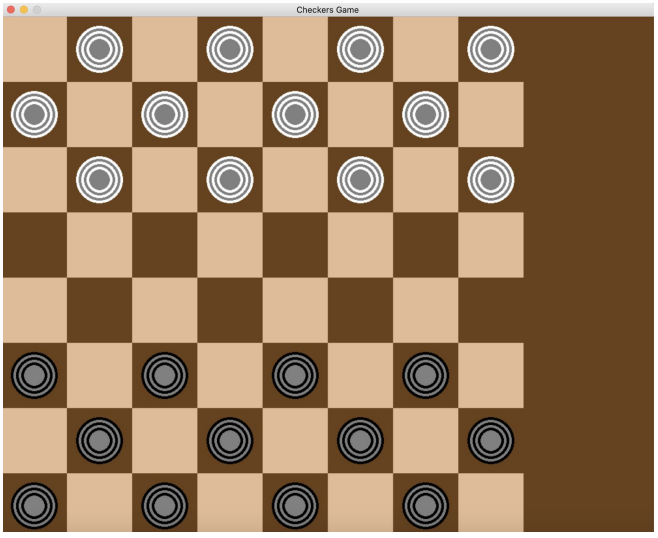
The implementation of the game is split into two parts: a menu and a main file. The menu file holds all the starting criteria such as the ai difficulty, the rules and the possibility of starting the game. The main file gathers all the game data from the checkers package and runs the game.

The files inside the checkers game include the following: piece, board, game, minimax, fixedvar and init.

The piece class is used for determining all the attributes of a piece. This includes its colour, its size, its position and whether it is a king or not.
The board class is used for displaying the board, retrieving data from it (pieces), calculating the possible moves each piece has and evaluating the possible moves for minimax
The game class is used to run the game and holds all the methods of updating the game. This includes moving the pieces on the board, updating displays (menu and game), gathering the inputs of the player, drawing possible moves and checking whether there is a winner.
The minimax file is used for the AI player to get all the valid moves and evaluate these accordingly by calling the board class.
The fixedvar file is used so that variables do not need to be repeatedly declared.

The purpose of this setup was so that the game class could use the methods of the piece and board objects to be able to play the game. Calling different boards allows the game class and the minimax file to see different possibilities of play.



# Program Functionality

## Game play

Interactive checkers gameplay (Human vs Computer)

Upon starting the game, the update (**line 50)** is called. This calls all the methods involved in drawing the board, the checkers, its pieces and declaring them objects. These methods include **draw_checkers, draw_piece, append_pieces** (**lines 300, 585, 357) .** When a player clicks on a piece, the board automatically displays the possible moves this piece can go by calling the **select** method **(line 156)**. This method uses the **generate_moves (401)** method to retrieve all the available moves by calling **_move_left** and **_move_right** (**lines 452, 497). Forced_capture** (**line 429)** comes into play here. While it already takes into account all valid moves (including forced capture), it decides whether there is another capturing piece at hand (**line 447)**. This allows the player to make a choice between which piece to take.

Different levels of verifiably effective AI cleverness, adjustable by the user

If the player clicks on an available checker square to move to, the game updates itself and displays the checker in its new position by using the **move** method**(line 324).** It is then the AIs turn. In the **Main** method **(line 37),** the AI is called as it is the white player's turn (**line 37)**. This uses the minimax file and method to determine all the possible moves it can move to. The depth used for the method is chosen on the start menu by the AI difficulty. This AI difficulty can be set by the user before entering the game. Depth is as followed, Easy = 2, Medium = 3, Hard = 4. This can be seen on **line 809**. The larger the depth, the deeper the minimax goes and thus the play will be more difficult.

The game will carry on until the game declares a winner. Jumps, kings, forced captures and regicide are all directly carried out. These will be explained further on.

## Search algorithm

State Representation, Successor function, Evaluation, Minimax pruning, heuristics

In the minimax method, if depth has not reached 0 and there is no winner (**line 634**), the method **get_all_moves (line 689)** is called to see every possible move a piece can perform at the current board (l**ine 700**) This current board is passed before as a parameter to the minimax method. To generate all these moves, the board method **generate_moves (line 401)** is inherited. Each of these moves is then simulated on new boards. These moves are passed back to the minimax method recursively until depth is 0 and a score is returned (**line 701)**. These scores are then compared to see which is the best possible move.

The heuristics used to obtain a good score are **(**method **evaluate (lines 268 - 298))** :

- How many pieces remain
    - Has a factor of 1
    - White pieces - black pieces
- How many king pieces remain
    - Has a factor of 2
    - White king pieces - black king pieces
- How many pieces are on the walls (safe spots)
    - Has a factor of 0.2
    - White wall pieces - black wall pieces

Alpha-Beta pruning is performed in the minimax method on line (**line 713)**. This allows the method to automatically break as there is no better move available out of the ones not yet searched.

**Validation of moves**

No invalid moves carried out by the AI

As previously mentioned the AI takes the same method the player does to generate the possible moves **(line 401)**. Like the player, they are forced to capture if possible and must remain within all bounds of the game.

Automatic check for valid user moves

This is checked for when generating the moves. Traversing along the diagonals is only possible. In terms of jumping another piece, (**line 196)** highlights this as the checker space must be empty and not contain a piece object.

Rejected of invalid user moves

All user moves which are invalid are rejected.

Forced capture

This has been discussed previously. In (**line 440),** if a move contains a jump, it is appended to the list of available captures. Moves which don't have a jump have an empty list. Only moves with jumps can then be carried out. The method **forced_capture (line 429)** determines whether there are multiple forced captures available and therefore gives the option for more than one move. **Draw_faded_moves (line 212)** then highlights the only available moves.

**Other features**

Multi-leg capturing moves for the user and AI

If the checker has the option to capture two pieces in a move, the **generate_moves(line 401)** method will show this as when using the method **move_left (line 452)** or **move_right (line 497)** to do this, if a piece can jump and still has a possible move, it will return a list with multiple entries. These entries will then be displayed when clicking on a piece to move that has the option to do so. The pieces which are jumped over are then removed by calling the **remove_piece (line 388)** method

King conversion at baseline (The king's row) as per the normal rules

In the move method in the board class, if the checker has reached either row 0 or row 7 on the board, **make_king (line 581)** is called which turns the piece into a king and displays it accordingly **(Line 605).**

Regicide - if a normal piece manages to capture a king, it is instantly crowned king and then the current turn ends.

The method move in the game class calls the move method in the board class which updates the display. If the piece jumped is a king as mentioned in (**line 203)**, the piece which jumped is made a king using the **make_king (line 581)** method. This can be seen in **(line 343)** where regicide is present.

Some kind of help feature that can be enabled at user's request to get hints about available moves, given the current game state. Sophisticated implementations of this feature may want to employ the AI to make suggestions on optimal moves.

**Human-Computer Interface**

Some kind of board representation displayed on screen

This is shown when the game has begun.

The interface properly updates the display after completed moves (User and AI moves)

The board class is called to update any UI displays

Fully interactive GUI that uses graphics

Menu, rules, AI difficulty, game display and winner are all displayed accordingly

Mouse interaction focus

The mouse clicks decide which checker to select and where to move. The arrow keys are used to navigate through the menu with user instructions given.

GUI pauses appropriately to show the intermediate steps of any multi-leg moves

A time delay is processed on any multi-leg moves to show the piece before it enters the final spot.

Dedicated display of the rules (e.g., a corresponding button opening a pop-up window)

Rules are shown on the menu page under the tab “rules”

# Appendix

```
1 import pygame
2 from checkers.fixedvar import space, white, surface
3 from checkers.game import Game
4 from checkers.minimax import minimax
5
6 # set the window of the game
7 window = pygame.display.set_mode((1000, 800))
8 # set the title
9 pygame.display.set_caption('Checkers Game')
10 FPS = 60
11 # set the surface
12 surface = surface
13
14
15 def main():
16     # method to run the game
17
18     # display menu
19     run = False
20     game = Game(window, surface)
21     while game.running:
22
23         game.curr_menu.display_menu()
24         game.game_loop()
25         if game.playing:
26             print(game.ai_difficulty)
27             run = True
28             break
29
30     clock = pygame.time.Clock()
31
32     # if run is True, the game should start
33     while run:
34         clock.tick(FPS)
35
36         # minimax set on the white colour
37         if game.turn == white:
```

```
38            value, new_board = minimax(game.get_board(), game.ai_difficulty + 1,
float('-inf'), float('inf'), white, game)
39            game.ai_turn(new_board)
40
41        for event in pygame.event.get():
42            if event.type == pygame.QUIT:
43                run = False
44            if event.type == pygame.MOUSEBUTTONDOWN:
45                pos = pygame.mouse.get_pos()
46                row, col = get_position(pos)
47                if row <8 and col <8:
48                    game.select(row, col)
49
50        game.update()
51
52    pygame.quit()
53
54 def get_position(pos):
55    # get the position of the mouse on the board
56    x, y = pos
57    row = y // space
58    col = x // space
59    return row, col
60
61 main()
62
63 --------------------
64
65 from .fixedvar import space
66 from checkers.board import Board, Piece
67 from menu import *
68 pygame.font.init
69
70
71 class Game:
72    # has all the functions for the gameplay
73    def __init__(self, window, surface):
74        pygame.init()
75        self.running, self.playing = True, False
76        self.ai_difficulty = 1  # initial AI difficulty
77        # initialise all keys to False
78        self.UP_KEY, self.DOWN_KEY, self.START_KEY, self.BACK_KEY,
self.RIGHT_KEY = False, False, False, False, False
79        self.display = pygame.Surface((width + 200,height))
```

```
80      self.font_name = pygame.font.get_default_font() # initial pygame font
81      self.main_menu = MainMenu(self)
82      self.options = RulesMenu(self)
83      self.curr_menu = self.main_menu
84      self.window = window
85      self._init()
86      self.surface = surface # used for displaying extra text
87      self.padding = 40
88      self.outline = 2
89
90  def update(self):
91      # updates the board
92      self.board.draw(self.window)
93      self.draw_faded_move(self.valid_moves, self.piece)
94      pygame.display.update()
95
96  def _init(self):
97      # initialise these variables
98      self.selected = None
99      self.board = Board()
100       self.turn = black
101       self.valid_moves = {}
102       self.piece = Piece(0, 0, black)
103
104   def game_loop(self):
105       # loop the game and choose between menu and actual game
106       # @return self.playing - to decide whether the game shall start
107       while self.playing:
108           self.input()
109           # if pressed, the game moves menu
110           if self.START_KEY:
111               self.playing = False
112           self.display.fill(black)
113           self.window.blit(self.display, (0,0))
114           pygame.display.update()
115           self.reset_keys()
116           return self.playing
117
118   def input(self):
119       # gets the input from the window to determine where the window should move to
120       for event in pygame.event.get():
121           if event.type == pygame.QUIT:
122               self.running, self.playing = False, False
123               self.curr_menu.run_display = False
```

```
124             if event.type == pygame.KEYDOWN:
125                 if event.key == pygame.K_RETURN:
126                     self.START_KEY = True
127                 if event.key == pygame.K_BACKSPACE:
128                     self.BACK_KEY = True
129                 if event.key == pygame.K_DOWN:
130                     self.DOWN_KEY = True
131                 if event.key == pygame.K_UP:
132                     self.UP_KEY = True
133                 if event.key == pygame.K_RIGHT:
134                     self.RIGHT_KEY = True
135
136     def reset_keys(self):
137         # resets the keys everytime after one is pressed
138         self.UP_KEY, self.DOWN_KEY, self.START_KEY, self.BACK_KEY,
self.RIGHT_KEY = False, False, False, False, False
139
140     def draw_text(self, text, size, x, y, colour = brown):
141         # draws the text for the menu
142         # @params text - the text chosen, size, x - x position, y - y position, colour
143         font = pygame.font.Font(self.font_name, size)
144         text_surface = font.render(text, True, colour)
145         text_rect = text_surface.get_rect()
146         text_rect.center = (x, y)
147         self.display.blit(text_surface, text_rect)
148
149     def winner(self):
150         # @return the board winner
151         return self.board.winner()
152
153     def reset(self):
154         self._init()
155
156     def select(self, row, col):
157         # select the row and column the player has chosen
158         # @params - the row and col of the selected spot
159         # return True if selection is possible. e.g if a piece is chosen, false otherwise, if out
of board,
160         # then no selection
161
162         # stay within the boundaries for selection
163         if row <=8 and col <= 8:
164             no_int = True
165             piece = self.board.get_piece(row, col)
```

```
166
167         # if piece is not a instance of int, e.g if it is an object and not nothing
168         if not isinstance(piece, int):
169             forced_cap, other_capture = self.board.forced_capture(piece)[0],
self.board.forced_capture(piece)[1]
170             no_int = False
171         if self.selected:
172             result = self._move(row, col)
173             if not result and not no_int:
174
175                 # if forced capture available select another piece is not available
176                 if forced_cap and not other_capture:
177                     self.selected = None
178                     self.select(row, col)
179
180         if piece != 0 and piece.colour == self.turn and forced_cap:
181             self.selected = piece
182             self.valid_moves, self.piece = self.board.generate_moves(piece)
183
184             return True
185
186         return False
187
188     else:
189         None
190
191   def _move(self, row, col):
192     # move the actual piece
193     # @params row, col
194     # @return true if piece is able to have moved
195     piece = self.board.get_piece(row, col)
196     if self.selected and piece == 0 and (row, col) in self.valid_moves:
197         self.board.move(self.selected, row, col)
198         piece_jumped = self.valid_moves[(row, col)]
199
200         # if the piece jumped is a king, the piece that jumped over automatically becomes
a king
201         if piece_jumped:
202             for skip in piece_jumped:
203                 if skip.king:
204                     self.board.move(self.selected, row, col, 1)
205             self.board.remove_piece(piece_jumped)
206         self.switch_turn()
207     else:
```

```
208             return False
209
210         return True
211
212     def draw_faded_move(self, moves, piece):
213         # draw the possible moves when a piece is selected
214         # @params moves - the potential moves a piece can do
215         # @params piece - the piece chosen to do the move
216
217         for move in moves:
218             row, col = move
219
220             # create a smaller checker in the proposed space with a highlighted box
221
222             self.window.blit(self.surface, (col * space + space // 2 - 50, row * space + space //
2 - 50))
223             radius = space // 2 - self.padding
224             pygame.draw.circle(self.window, piece.colour, (col * space + space // 2, row *
space + space // 2), radius + 24 * 0.5 + self.outline)
225             pygame.draw.circle(self.window, grey, (col * space + space // 2, row * space +
space // 2), radius + 20 * 0.5 + self.outline)
226             pygame.draw.circle(self.window, piece.colour, (col * space + space // 2, row *
space + space // 2), radius + 16 * 0.5 + self.outline)
227             pygame.draw.circle(self.window, grey, (col * space + space // 2, row * space +
space // 2), radius + 12 * 0.5 + self.outline)
228             pygame.draw.circle(self.window, piece.colour, (col * space + space // 2, row *
space + space // 2), radius + 8 * 0.5 + self.outline)
229             pygame.draw.circle(self.window, grey, (col * space + space // 2, row * space +
space // 2), radius + 4 * 0.5 + self.outline)
230             pygame.draw.circle(self.window, grey, (col * space + space // 2, row * space +
space // 2), radius + self.outline)
231
232     def switch_turn(self):
233         # change the turn of the game
234         self.valid_moves = {}
235         if self.turn == black:
236             self.turn = white
237         else:
238             self.turn = black
239
240     def get_board(self):
241         #@return the board object
242         return self.board
243
```

```
244     def ai_turn(self, board):
245         # ai moves
246         # @params board object
247         self.board = board
248         self.switch_turn()
249 
250 ------------------------
251 
252 import pygame
253 from checkers.fixedvar import brown, black, white, space, columns, rows, dark_brown,
black_winner, white_winner
254 from .piece import Piece
255 
256 
257 class Board:
258     # This board holds all the functions for drawing, amending and evaluating the board
and the pieces inside
259 
260     def __init__(self):
261         self.board = []
262         self.black_pieces = 12  # number of black pieces remaining
263         self.white_pieces = 12  # number of white pieces remaining
264         self.black_kings = 0  # number of black king pieces remaining
265         self.white_kings = 0  # number of white king pieces remaining
266         self.append_pieces()  # creates board object
267 
268     def evaluate(self):
269         # Evaluates the Board, used for the Minimax function
270         # @return score = the evaluation score
271 
272         column_pieces_w = 0
273         column_pieces_b = 0
274         end_pieces = 0.25
275         white_checkers = self.iterpieces(white)
276         for row in self.board:
277             for piece in white_checkers:
278                 if row[0] == piece or row[7] == piece:
279                     column_pieces_w += 0.2
280 
281         black_checkers = self.iterpieces(black)
282         for row in self.board:
283             for piece in black_checkers:
284                 if row[0] == piece or row[7] == piece:
285                     column_pieces_b += 0.2
```

```
286
287         # How many pieces are on the wall
288         wall_score = column_pieces_w - column_pieces_b
289
290         # How many kings remain
291         king_score = (self.white_kings - self.black_kings) * 2
292
293         # How many pieces remain
294         remaining_pieces = self.white_pieces - self.black_pieces
295
296         score = remaining_pieces + king_score + wall_score
297
298         return score
299
300     def draw_checkers(self, window):
301         # Draws the Checkers Board
302         # @params window - this is the window itself where the squares are drawn onto
303         window.fill(dark_brown)
304
305         # loop through rows and columns to generate each square
306         for row in range(rows):
307             for col in range(row % 2, rows, 2):
308                 pygame.draw.rect(window, brown, (row * space, col * space, space, space))
309
310     def iterpieces(self, colour):
311         # Goes through all the pieces and returns all pieces of a colour
312         # @params colour - this is used to determine which colour is present
313         # @return - returns the pieces
314         pieces = []
315
316         # iterates through the board
317         for row in self.board:
318             for piece in row:
319                 # Piece is present if piece returns an object and not 0
320                 if piece != 0 and piece.colour == colour:
321                     pieces.append(piece)
322         return pieces
323
324     def move(self, piece, row, col, regicide=0):
325         # move the actual piece with the selected square
326         # @params piece - the piece chosen
327         # @params row - the row where the piece will move to
328         # @params col - the col where the piece will move to
```

```
329         # @params regicide - if regicide = 1, the piece will automatically become a king
upon its move
330
331         self.board[piece.row][piece.col], self.board[row][col] = self.board[row][col],
self.board[piece.row][piece.col]
332         piece.move(row, col)
333
334         # check if piece has reached the final row to determine if it is a king or not
335         if row == 7 or row == 0:
336             piece.make_king()
337             if piece.colour == black:
338                 self.black_kings += 1
339             else:
340                 self.white_kings += 1
341
342         # make king if regicide carried out
343         if regicide == 1:
344             piece.make_king()
345
346     def get_piece(self, row, col):
347         # get one piece
348         # @params row - the row
349         # @params col - the col
350         # @return piece object
351
352         # stay within the limits as side of window can be theoretically selected
353         if row <= 8 and col <= 8:
354             return self.board[row][col]
355
356
357     def append_pieces(self):
358         # create the board - add each piece to a valid starting square, add object not drawing
359         for row in range(rows):
360             self.board.append([])
361             for col in range(columns):
362                 if col % 2 == ((row + 1) % 2):
363                     if row < 3:
364                         self.board[row].append(Piece(row, col, white))
365                     elif row > 4:
366                         self.board[row].append(Piece(row, col, black))
367                     else:
368                         self.board[row].append(0)
369                 else:
370                     self.board[row].append(0)
```

```
371
372     def draw(self, window):
373         # draw the actual piece object onto the board by calling piece.draw_piece function
374         # @params - window - the window
375         self.draw_checkers(window)
376         for row in range(rows):
377             for col in range(columns):
378                 piece = self.board[row][col]
379                 if piece != 0:
380                     piece.draw_piece(window)
381
382         # See if a player has won - if so display the images on the side of the window
383         if self.white_pieces == 0:
384             window.blit(black_winner, (500, 0))
385         if self.black_pieces == 0:
386             window.blit(white_winner, (500, 0))
387
388     def remove_piece(self, pieces):
389         # remove the pieces from the board if jumped
390         # @params pieces - the pieces that have been jumped
391
392         # multiple incase a multi-leg jump was made
393         for piece in pieces:
394             self.board[piece.row][piece.col] = 0
395             if piece != 0:
396                 if piece.colour == black:
397                     self.black_pieces -= 1
398                 else:
399                     self.white_pieces -= 1
400
401     def generate_moves(self, piece):
402         # generate all possible moves that a piece may do. Includes forced capture.
403         # @params piece - piece object chosen
404         # @return all possible moves in a dictionary, and the object moved
405
406         possible_moves = {}
407         left = piece.col - 1
408         right = piece.col + 1
409         row = piece.row
410
411         if piece.colour == black or piece.king:
412             possible_moves.update(self._move_left(row - 1, max(row - 3, -1), -1,
piece.colour, left))
```

```
413             possible_moves.update(self._move_right(row - 1, max(row - 3, -1), -1,
piece.colour, right))
414
415         if piece.colour == white or piece.king:
416             possible_moves.update(self._move_left(row + 1, min(row + 3, rows), 1,
piece.colour, left))
417             possible_moves.update(self._move_right(row + 1, min(row + 3, rows), 1,
piece.colour, right))
418
419         # loops through all the available moves, if a forced capture is possible, only give
this option
420         move = {}
421         for init_pos, end_pos in possible_moves.items():
422             if len(end_pos) > 0:
423                 move[init_pos] = end_pos
424         if len(move) != 0:
425             possible_moves = move
426
427         return possible_moves, piece
428
429     def forced_capture(self, piece):
430         # This method takes the possible moves and decides if a forced capture is possible
431         # @pararms - a piece object
432         # @return forced cap - the forced capture of the piece
433         # @return other_forced_cap - there is another forced capture avaialble
434         all_pieces = self.iterpieces(piece.colour)
435         forced_cap = True
436         other_forced_cap = False
437         current_piece_capture = False
438         for temp_piece in all_pieces:
439             valid_moves = self.generate_moves(temp_piece)
440             for move, capture_list in valid_moves[0].items():
441                 if len(capture_list) > 0:
442                     if temp_piece.row != piece.row or temp_piece.col != piece.col:
443                         other_forced_cap = True
444                     if temp_piece.row == piece.row and temp_piece.col == piece.col:
445                         current_piece_capture = True
446
447         if other_forced_cap and not current_piece_capture:
448             forced_cap = False
449
450         return forced_cap, other_forced_cap
451
452     def _move_left(self, init, end, step, colour, left, jumped=[]):
```

```
453         # This method takes a piece and generates the moves to the left of it diagonally
454         # @pararms - init - start position
455         # @pararms - end - stop position
456         # @pararms - step - if jumped over
457         # @pararms - colour - colour of piece
458         # @pararms - left - moving left
459         # @pararms - jumped - if a piece has been jumped - used for multi-leg
460         # @return moves - the available moves to the left
461
462         moves = {}
463         last = []
464
465         # check if within bounds
466         for r in range(init, end, step):
467             if left < 0:
468                 break
469
470             # get current position
471             current = self.board[r][left]
472             if current == 0:
473                 if jumped and not last:
474                     break
475                 elif jumped:
476                     moves[(r, left)] = last + jumped
477                 else:
478                     moves[(r, left)] = last
479
480                 if last:
481                     if step == -1:
482                         row = max(r - 3, -1)
483                     else:
484                         row = min(r + 3, rows)
485                     moves.update(self._move_left(r + step, row, step, colour, left - 1,
jumped=last))
486                     moves.update(self._move_right(r + step, row, step, colour, left + 1,
jumped=last))
487                 break
488             elif current.colour == colour:
489                 break
490             else:
491                 last = [current]
492
493             left -= 1
494
```

```
495         return moves
496
497     def _move_right(self, init, end, step, colour, right, jumped=[]):
498         # This method takes a piece and generates the moves to the left of it diagonally
499         # @pararms - init - start position
500         # @pararms - end - end position
501         # @pararms - step - if jumped over
502         # @pararms - colour - colour of piece
503         # @pararms - left - moving right
504         # @pararms - jumped - if a piece has been jumped - used for multi-leg
505         # @return moves - the available moves to the right
506         moves = {}
507         last = []
508
509         # check if within bounds
510         for r in range(init, end, step):
511             if right >= columns:
512                 break
513
514             # get current position
515             current = self.board[r][right]
516             if current == 0:
517                 if jumped and not last:
518                     break
519                 elif jumped:
520                     moves[(r, right)] = last + jumped
521                 else:
522                     moves[(r, right)] = last
523
524                 if last:
525                     if step == -1:
526                         row = max(r - 3, -1)
527                     else:
528                         row = min(r + 3, rows)
529                     moves.update(self._move_left(r + step, row, step, colour, right - 1,
jumped=last))
530                     moves.update(self._move_right(r + step, row, step, colour, right + 1,
jumped=last))
531                 break
532             elif current.colour == colour:
533                 break
534             else:
535                 last = [current]
536
```

```
537             right += 1
538
539         return moves
540
541     def winner(self):
542         # determines if there is a winner or not
543         # @return the colour of the winner - if no winner - returns none
544         if self.black_pieces <= 0:
545             return white
546         if self.white_pieces <= 0:
547             return black
548         return None
549
550 ------------------------
551
552 import pygame
553 from .fixedvar import black, space, grey, king, black_winner
554 # piece class
555
556 class Piece:
557     #class handles all the piece object functions
558     def __init__(self, row, col, colour):
559         self.padding = 40
560         self.outline = 2
561         self.row = row
562         self.col = col
563         self.colour = colour
564         self.king = False
565
566         # used for determining the direction a piece can move
567         if self.colour == black:
568             self.direction = -1
569         else:
570             self.direction = 1
571
572         self.x = 0
573         self.y = 0
574         self.piece_position()
575
576     def piece_position(self):
577         # determine the piece position
578         self.x = space * self.col + space // 2
579         self.y = space * self.row + space // 2
580
```

```
581     def make_king(self):
582         # make the piece a king
583         self.king = True
584 
585     def draw_piece(self, window, n=1, hint=0):
586         # draw the actual piece in the window
587         # @params window - the window
588         # @params n - a int to determine how smaller the circle radius should go - visual
purposes
589 
590         # draw the piece
591         radius = space // 2 - self.padding
592         pygame.draw.circle(window, self.colour, (self.x, self.y), radius + 24 * n +
self.outline)
593         pygame.draw.circle(window, grey, (self.x, self.y), radius + 20 * n + self.outline)
594         pygame.draw.circle(window, self.colour, (self.x, self.y), radius + 16 * n+
self.outline)
595         pygame.draw.circle(window, grey, (self.x, self.y), radius + 12* n + self.outline)
596         pygame.draw.circle(window, self.colour, (self.x, self.y), radius + 8 * n+
self.outline)
597 
598         # if not king, draw extra circles
599         if not self.king:
600             pygame.draw.circle(window, grey, (self.x, self.y), radius + 4 * n + self.outline)
601             pygame.draw.circle(window, grey, (self.x, self.y), radius + self.outline)
602 
603         # if king, draw the image on top
604         if self.king:
605             window.blit(king, (self.x - king.get_width() // 2, self.y - king.get_height() // 2))
606 
607 
608 
609     def move(self, row, col):
610         # method to move the piece to that row, col and calc the position afterwards
611         self.row = row
612         self.col = col
613         self.piece_position()
614 
615     def __repr__self(self):
616         return str(self.colour)
617 
618 ----------------------
619 
620 from copy import deepcopy
```

```
621 import pygame
622 from .fixedvar import black, white
623
624 def minimax(curr_board, depth, alpha, beta, max_player, game):
625     # method to perform the minimax function
626     # @params curr_board - the board position
627     # @params depth - the depth chosen for the evaluation to go through
628     # @params alpha
629     # @params beta
630     # @params max_player - the player wanting to get the best move
631     # @params game - the current game
632     # @return the evaluation score and the best move for the player chosen
633
634     if depth == 0 or curr_board.winner() is not None:
635         return curr_board.evaluate(), curr_board
636
637     if max_player:
638         # set the max player to - infinity
639         maxEval = float('-inf')
640         best_move = None
641         # iterate through all moves recursively
642         for move in get_all_moves(curr_board, white, game):
643             evaluation = minimax(move, depth-1, alpha, beta, False, game)[0]
644             maxEval = max(maxEval, evaluation)
645             alpha = max(alpha, evaluation)
646             # perform pruning
647             if beta <= alpha:
648                 break
649             # if score is better, change best move
650             if maxEval == evaluation:
651                 best_move = move
652
653         return maxEval, best_move
654     else:
655         # set the min player to + infinity
656         minEval = float('inf')
657         best_move = None
658         # iterate through all moves recursively
659         for move in get_all_moves(curr_board, black, game):
660             evaluation = minimax(move, depth-1, alpha, beta, True, game)[0]
661             minEval = min(minEval, evaluation)
662             beta = min(beta, evaluation)
663             # perform pruning
664             if beta <= alpha:
```

```
665                 break
666             # if score is better, change best move
667             if minEval == evaluation:
668                 best_move = move
669         return minEval, best_move
670 
671 def simulate_move(piece, move, board, jumped_piece):
672     # simulate move method to see where the piece would go and what it would entail
673     # @params piece - the piece to move
674     # @params move - the move itself
675     # @params board - the board position
676     # @params jumped_piece - if a piece was jumped or not
677     # @return the board to see the "possible" position
678     board.move(piece, move[0], move[1])
679     if jumped_piece:
680         for skip in jumped_piece:
681             # perform regicide
682             if skip.king:
683                 board.move(piece, move[0], move[1], 1)
684         board.remove_piece(jumped_piece)
685 
686     return board
687 
688 
689 def get_all_moves(board, colour, game):
690     # get all the moves from the board
691     # @params piece - the piece to move
692     # @params colour - is it black or white which all moves are being retrieved for
693     # @return the moves possible for all pieces
694     moves = []
695     for piece in board.iterpieces(colour):
696         # Piece is not available for AI if forced cap is available
697         if not board.forced_capture(piece)[0]:
698             continue
699         valid_moves = board.generate_moves(piece)
700         for move, skip in valid_moves[0].items():
701             # make a deepcopy of the board
702             temp_board = deepcopy(board)
703             # temporary copies are made of the pieces in the board to see the possible moves
704             temp_piece = temp_board.get_piece(piece.row, piece.col)
705             new_board = simulate_move(temp_piece, move, temp_board, skip)
706             moves.append(new_board)
707     return moves
708 
```

```
709 ----------------------
710
711 import pygame
712 from checkers.fixedvar import width, height, black, white, brown, dark_brown,
light_brown, grey
713
714 #file for UI before game
715
716 class Menu():
717     # main class used by mainmenu and rulesmenu to inherit.
718     def __init__(self, game):
719         self.width, self.height = 1000, 800
720         self.game = game
721         self.mid_w, self.mid_h = self.width / 2, self.height / 2
722         self.run_display = True
723         self.menu_cursor = pygame.Rect(0, 0, 50, 50)
724         self.ai_cursor = pygame.Rect(0, 0, 50, 50)
725         self.remove = - 90
726
727     def draw_menu_cursor(self):
728         # draw the asterisks symbol next to the main menu options
729         self.game.draw_text('*', 50, self.menu_cursor.x, self.menu_cursor.y - 50)
730
731     def draw_AI_cursor(self):
732         # draw the underscore symbol next to the AI options
733         self.game.draw_text('_', 50, self.ai_cursor.x, self.ai_cursor.y - 65)
734
735     def update_screen(self):
736         # update the screen after each key is pressed
737         self.game.window.blit(self.game.display, (0, 0))
738         pygame.display.update()
739         self.game.reset_keys()
740
741
742 class MainMenu(Menu):
743     # main menu class - has all the variables for the starting screen- UI
744     def __init__(self, game):
745         Menu.__init__(self, game)
746         self.state = "Start"
747         self.startx, self.starty = self.mid_w, self.mid_h + 20
748         self.rulesx, self.rulesy = self.mid_w, self.mid_h + 80
749         self.aiwx, self.aiwy = self.mid_w, self.mid_h + 200
750
751         # get the AI positions (easy, medium, hard)
```

```
752         self.ai1x, self.ai1y = self.mid_w - 250, self.mid_h + 250,
753         self.ai2x, self.ai2y = self.mid_w, self.mid_h + 250,
754         self.ai3x, self.ai3y = self.mid_w + 250, self.mid_h + 250,
755
756         # get the positions for the cursors
757         self.menu_cursor.midtop = (self.startx - 90, self.starty + 10)
758         self.ai_cursor.midtop = (self.ai1x + 25, self.ai1y + 10)
759
760         # initialise AI level to 1
761         self.ai_level = 1
762
763     def display_menu(self):
764         # displays the starting screen
765         self.run_display = True
766         while self.run_display:
767             self.game.input()
768             self.check_input()
769             self.game.display.fill(dark_brown)
770             self.game.draw_text('Checkers Game', 100, self.width / 2, self.height / 2 - 200)
771             self.game.draw_text("Start Game", 30, self.startx, self.starty - 50)
772             self.game.draw_text("Rules", 30, self.rulesx, self.rulesy - 50)
773             self.game.draw_text("Choose Your AI difficulty:", 20, self.aiwx, self.aiwy - 50)
774             self.game.draw_text("Easy", 20, self.ai1x, self.ai1y - 50)
775             self.game.draw_text("Medium", 20, self.ai2x, self.ai2y - 50)
776             self.game.draw_text("Hard", 20, self.ai3x, self.ai3y - 50)
777             self.game.draw_text("Use arrow keys : DOWN, RIGHT : to move between
options", 15, self.ai2x, self.ai2y + 50,
778                                 light_brown)
779             self.game.draw_text("Use key : ENTER : to choose menu option", 15, self.ai2x,
self.ai2y + 80, light_brown)
780             self.draw_menu_cursor()
781             self.draw_AI_cursor()
782             self.update_screen()
783
784     def move_cursor(self):
785         # move the cursor
786         if self.game.DOWN_KEY:
787
788             if self.state == 'Start':
789                 self.menu_cursor.midtop = (self.rulesx + self.remove, self.rulesy + 10)
790                 self.state = 'Options'
791             elif self.state == 'Options':
792                 self.menu_cursor.midtop = (self.startx + self.remove, self.starty + 10)
793                 self.state = 'Start'
```

```
794         elif self.game.RIGHT_KEY:
795
796             if self.ai_level == 1:
797                 self.ai_cursor.midtop = (self.ai2x + 25, self.ai2y + 10)
798                 self.ai_level = 2
799             elif self.ai_level == 2:
800                 self.ai_cursor.midtop = (self.ai3x + 25, self.ai3y + 10)
801                 self.ai_level = 3
802             elif self.ai_level == 3:
803                 self.ai_cursor.midtop = (self.ai1x + 25, self.ai1y + 10)
804                 self.ai_level = 1
805
806     def check_input(self):
807         # check what was entered to see whether the game should start or the options menu
should be displayed
808         self.move_cursor()
809         self.game.ai_difficulty = self.ai_level
810         if self.game.START_KEY:
811             if self.state == 'Start':
812                 self.game.playing = True
813             elif self.state == 'Options':
814                 self.game.curr_menu = self.game.options
815             self.run_display = False
816
817
818 class RulesMenu(Menu):
819     def __init__(self, game):
820         Menu.__init__(self, game)
821         self.x, self.y = self.width / 2, self.height / 10
822         self.movedown = 40
823
824     def display_menu(self):
825         #draw the rules menu
826         self.run_display = True
827         while self.run_display:
828             self.game.input()
829             self.check_input()
830             self.game.display.fill(dark_brown)
831             self.game.draw_text("Basic Gameplay", 30, self.x, self.y - 25, grey)
832             self.game.draw_text("The human player controls the black pieces while the AI
player controls the white.",
833                                 20, self.x, self.y + self.movedown - 25)
834             self.game.draw_text("Pieces may only move diagonally forwards.", 20, self.x,
835                                 self.y + self.movedown * 2 - 25)
```

```
836         self.game.draw_text("Capturing Pieces", 30, self.x, self.y + self.movedown * 3,
grey)
837         self.game.draw_text("Pieces can only move one square in a non-capturing
move.", 20, self.x,
838                             self.y + self.movedown * 4)
839         self.game.draw_text("To capture an opponent's piece, a player must jump over
their opponents piece", 20,
840                             self.x, self.y + self.movedown * 5)
841         self.game.draw_text("into an empty square. The captured piece is then removed.
", 20, self.x,
842                             self.y + self.movedown * 6 - 15)
843         self.game.draw_text("A piece may make multiple captures in one move if
multiple valid jumps can be made.",
844                             20, self.x, self.y + self.movedown * 7 - 25)
845         self.game.draw_text("A piece is forced to capture an opponent's piece if they are
able to.", 20, self.x,
846                             self.y + self.movedown * 8 - 25)
847         self.game.draw_text("King Piece", 30, self.x, self.y + self.movedown * 9, grey)
848         self.game.draw_text(
849             "If a player's piece reaches the last row on their opponent's side, the piece is
made a king.", 20,
850             self.x, self.y + self.movedown * 10)
851         self.game.draw_text("King pieces may move diagonally forwards and
backwards.", 20, self.x,
852                             self.y + self.movedown * 11)
853         self.game.draw_text("If a non-king piece captures a king piece, that piece is
automatically a king.", 20,
854                             self.x, self.y + self.movedown * 12)
855         self.game.draw_text("Game Objective", 30, self.x, self.y + self.movedown * 13 +
25, grey)
856         self.game.draw_text("A player wins when they have captured all their opponent's
pieces.", 20, self.x,
857                             self.y + self.movedown * 14 + 25)
858         self.game.draw_text("Use key : BACKSPACE : to go back to the menu", 15, 500,
750, )
859         self.update_screen()
860
861     def check_input(self):
862         #check the input to see whether to go back to the main menu
863         if self.game.BACK_KEY:
864             self.game.curr_menu = self.game.main_menu
865             self.run_display = False
866
867 --------------------
```

```
868
869 import pygame
870
871
872 surface = pygame.Surface((100,100))
873 surface.set_alpha(100)
874
875 width = 800
876 height = 800
877 rows = 8
878 columns = 8
879
880 # King Image
881 king = pygame.transform.scale(pygame.image.load("assets/king.png"), (35, 25))
882 white_winner =
pygame.transform.scale(pygame.image.load("assets/whitehaswon.png"),(800,500))
883 black_winner =
pygame.transform.scale(pygame.image.load("assets/blackhaswon.png"),(800,500))
884
885 # Colour codes for UI
886 red = (255, 0, 0)
887 white = (255, 255, 255)
888 black = (0, 0, 0)
889 brown = (222, 188, 153)
890 light_brown = (200, 208, 168)
891 dark_brown = (101, 67, 33)
892 grey = (128, 128, 128)
893
894 surface.fill(white)
895 space = int(width / columns)
896
```