Draughts Game Using a Minimax algorithm

Rohit Pai
Candidate Number: 198771
University of Sussex

December 9, 2021

Contents

Abstract

1 Introduction

The task was to build a draughts game which had a GUI and that could be played against an AI with varying difficulty levels. Originally I did not have a good idea on how to start the building the game. I found a tutorial series which covered the basic game mechanics and class structure [4]. To then understand how to develop the minimax algorithm I looked at Sebastian Leagues video [3] and the part 2 series of the draughts tutorial series [1] was used as a guide. Although YouTube tutorial videos have been used, lots of the code have been adapted to meet the marking criteria. I have used python to create the game as python is non typed language meaning it should be easier to make a game like this without having to deal type errors.

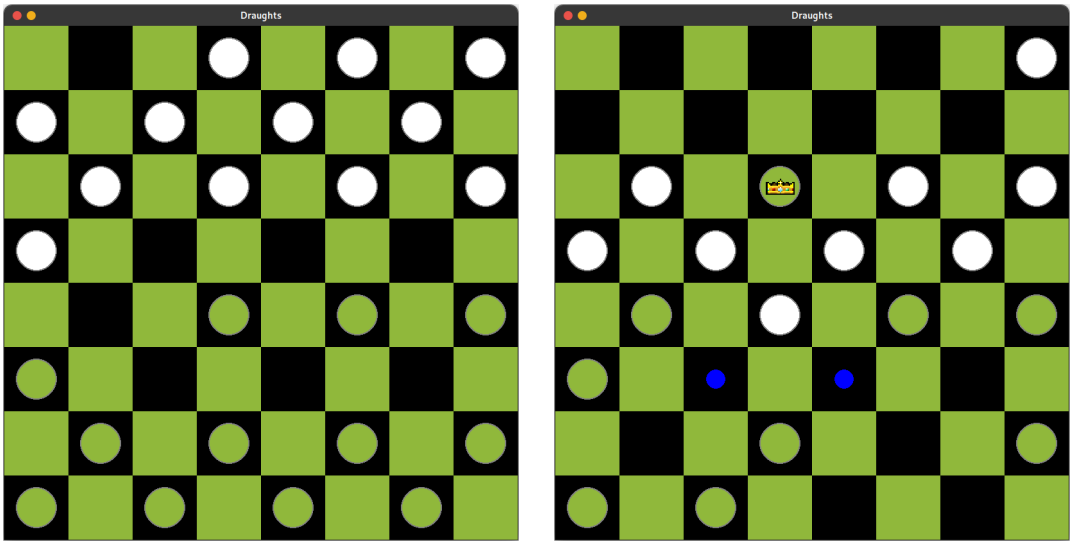
I have created 5 classes which house the different functions to make the game work. The board class keeps a track of the board and the board state. The piece class keeps track of an individual piece and its states e.g. what colour and whether it is king or not. The game manager class keeps track of the whole game and drawing the actual GUI on the screen. The minimax class houses the minimax method and it's respective helper methods. The main.py file holds the main menu loop and the main game loop and 2 helper functions. The constants.py file holds static variables i.e. the variables that are not going to be changed at all. The main menu was created with the help of this video [2]. The general structure of the menu design was taken from the video and then modified to fit the draughts game.

2 Description of program functionality

2.1 Gameplay

2.1.1 Interactive checkers gameplay (Human vs. Computer) of some sort

See figures below 1.



(a) Main game board with the pieces

(b) Main game with a green king

Figure 1: Gameplay

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

To select a difficulty level, either easy medium or hard have to be selected in the main menu. After the difficulty level is selected the game loads up and the user can start playing the game.

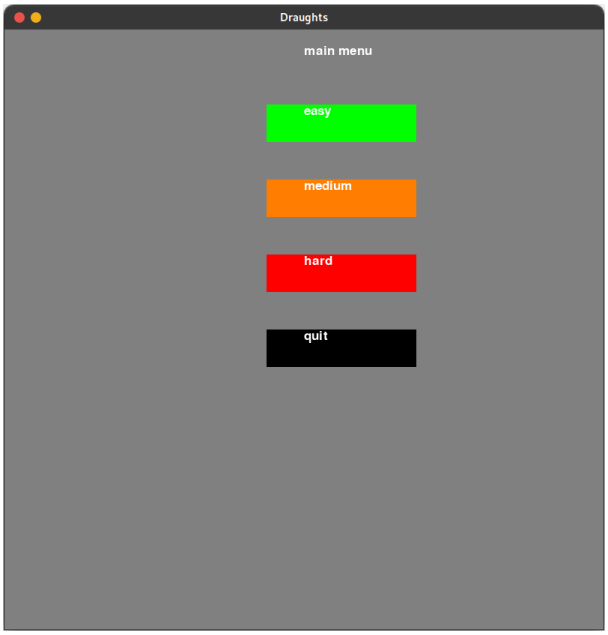


Figure 2: In game menu system

2.2 Search algorithm

2.2.1 Appropriate and efficient state representation

The main game has 4 states that it can be in. The initial state is when the game starts and all the pieces are the beginning and it's the players turn to move but the player has not started to move yet. See figure 3. The next state is when the player has selected a piece to move but has not moved

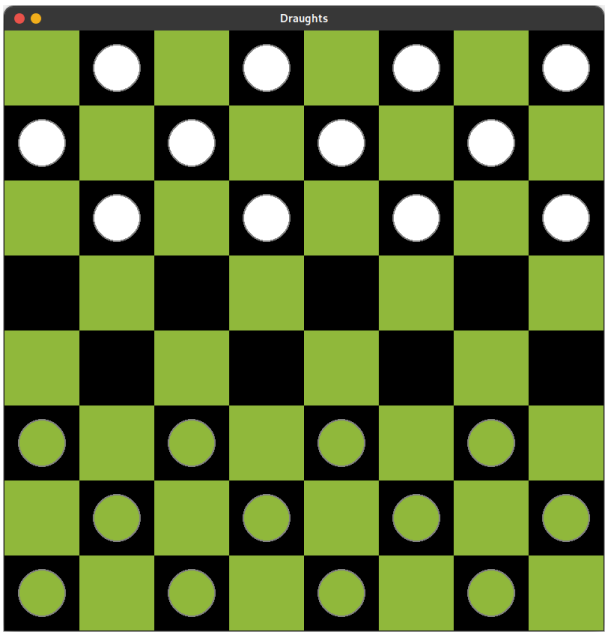


Figure 3: Initial state with the player's turn at hand and no move has been made yet.

yet. See figure 4. The next state is when the AI needs to move. Since the AI is instantaneous, this

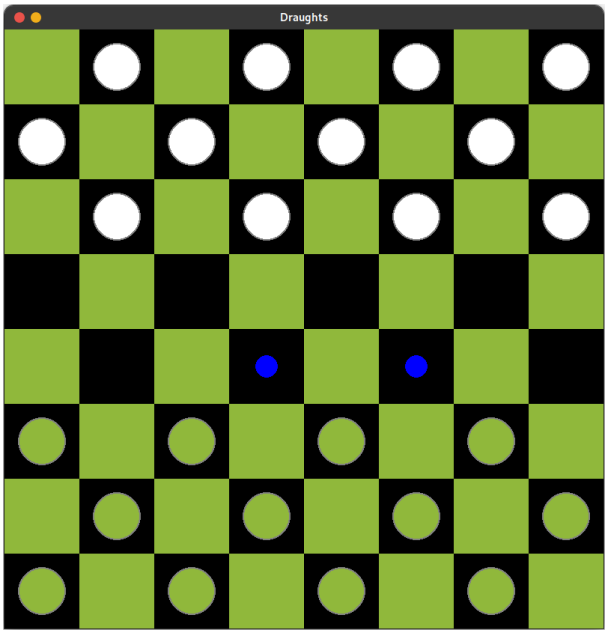


Figure 4: The player has selected a piece to move but not moved yet. The available moves are shown in blue

state cannot be seen unless the depth of the AI is set to a very high number, where the computer hardware needs time to compute the result. The final state is when the player or AI has won the game, this can be found out by calling the winner function in the game manager class which calls the winner function in the board class. This function checks to see if there are no more green or white pieces on the board and returns the piece that has won.

2.2.2 Reasonable successor function to generate AI moves

The implementation of minimax in this game accounts for a successor to generate AI moves. See minimax class and the AI function inside it in the appendix. The function accounts for the successor based on the number of pieces left on the board. It has a higher chance of trying to become king if the AI benefits from it i.e. remove more player pieces from the board and obtain more king pieces for the AI.

2.2.3 Minimax evaluation

In the minimax implementation, the current move that the AI or player can make is being evaluated At the beginning, the AI assumes the player has the upper hand and tries to find a move which works in favour of the AI. The algorithm keeps trying to find the best possible move for all pieces and then compares those best moves to get a single best move. It evaluates until the depth limit is reached. When the limit has been reached, the score of the board is calculated and returned alongside the modified board with the moved piece on it.

2.2.4 Appropriate use of heuristics

The AI wants to get rid of more player pieces. The minimax algorithm returns the result of the function scoreOfTheBoard and the modified board. When the AI function inside the minimax class runs, it runs recursively until the depth has reached 0 and then the score of the board is calculated. The score of the board returns a king incentivised value either positive or negative. This is returned back via the recursive nature and then evaluated against current best move evaluation.

2.3 Validation of moves

2.3.1 No invalid moves carried out by the AI

During the minimax algorithm getValidMoves function is run on each of the pieces for the colour, in this case white. What getValidMoves, in the board class, calculates is all the valid moves for a given piece based on the rest of the pieces on the board until it finds a space on the board where there

are two blank spaces i.e. spaces that are set as None. It then stores these into a dictionary with the keys being the row and column as a tuple and values being the list of jump-able pieces. Then the AI would use this to calculate what's best move the AI.

2.3.2 Automatic check for valid user moves

The same function getValidMoves, in the board class, is also run on the player which returns the same dictionary but the player would decide which move to make. See figure 4. During getValidMoves, both the left and right moves are calculated and then returned for the given jump and piece.

2.3.3 Rejection of invalid user moves

Since getValidMoves, in the board class, returns all the available moves on a specific piece and the board shows what the available moves are, the player can not then select a move that is not within the available moves. After selected a piece, the available moves are shown in blue, see figure 4, the player can then choose one of the available moves. If the player selects a space on the board, the space is then automatically checked to see if it is contained within available moves, if it is, move the player to the new space otherwise it does nothing, allowing the user to select again. The game will not move the piece until the player has selected one of the valid moves.

2.3.4 Forced Capture

When you select a piece, getValidMoves is run and then the valid moves is returned. Then in the move function the valid moves are filtered out to either include all the moves meaning none of them can capture anything, or include all the moves that can capture an AI piece. See figure 5.

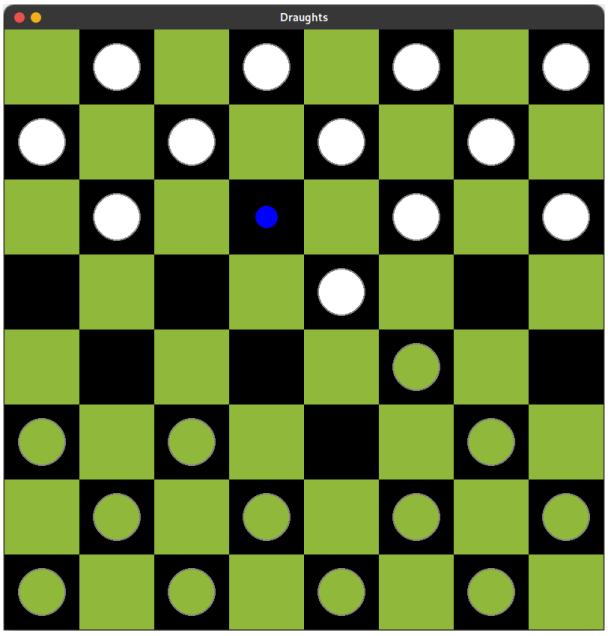


Figure 5: Only the capturable moves are shown

2.4 Other Features

2.4.1 Multi-leg capturing moves for the user

When the player has the ability to complete a multi-leg move the blue circles will be shown in the respective spaces on the board. See figure 6. The player can then select the moves within the multi-leg capture to show intermediate steps or the final position to jump straight there and complete a multi-leg jump

2.4.2 Multi-leg capturing moves for the AI

When the AI has the ability to complete a multi-leg move, it will favour this move over other moves because the function scoreOfTheBoard will evaluate the move and it will consider it better for the AI since more player pieces have been removed from the board in the process.

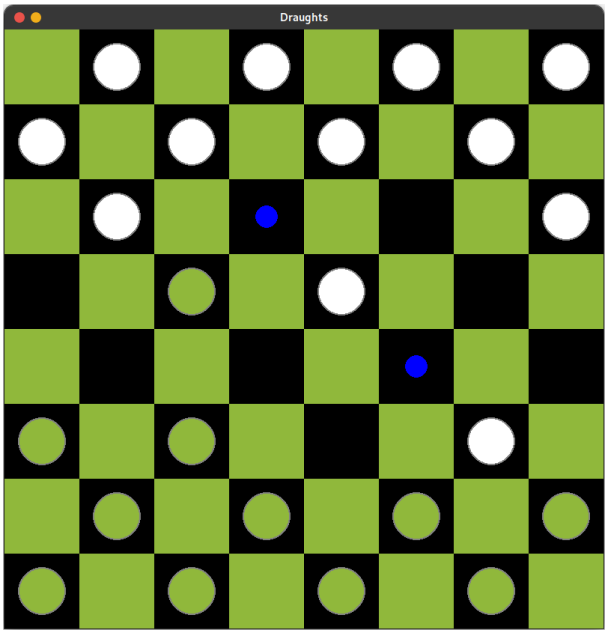


Figure 6: Multi-leg capturing represented by the blue circles on the board.

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

When a player piece, a green piece, or an AI piece, white piece, has reached the opposite sides of the board, the current piece is then converted to a king. See figure 1b The piece class has a boolean to denote whether a piece is king or not and then the piece is then rendered based on that. If the piece is king it can move in all four directions. In the getValidMoves function the two if statements check whether a piece is the given colour or king. If it is king, it checks both left and right of both up and down movements.

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

When the player jumps over a king, the player is then made king. This is all completed in the move function. In the function, the current jumped over piece is checked to see if it is king and if it is then the player or AI piece is then converted to a king and then move then continues as normal.

2.4.5 Some kind of help feature that can be enabled at the user's request to get hints about available moves, given the current game state.

When a player selects a piece to move, getValidMoves is run, this then calculates the given valid moves for a piece. By default the valid moves are then drawn on the board as blue circles using the drawValidMoves function. This function then draws a circle at the position of the valid move and then when a piece is moved the board is updated and then drawn valid moves get removed.

2.5 Some kind of board representation displayed on screen

See all the figures in this report.

2.6 Theinterface properly updates the display after completed moves (User and AI moves)

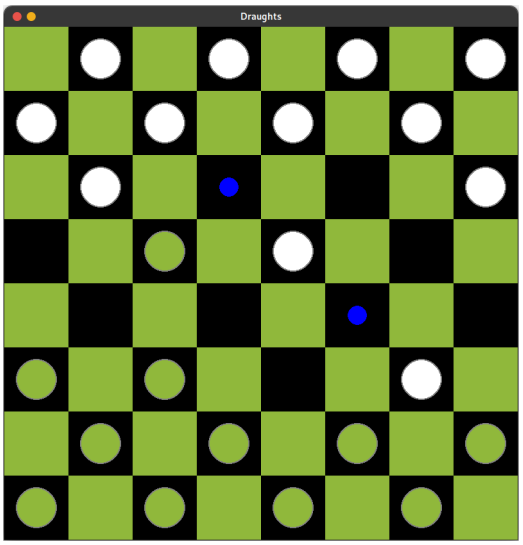
The main game loop runs using a clock timer and runs every 60 seconds, meaning it can be thought of like a game that runs at 60FPS. Which means the board then gets updated every frame. Once a turn is fully finished, the gane runs the swapTurn function w

2.7 Mouse interaction focus, e.g., click to select & click to place, or drag & drop (better)

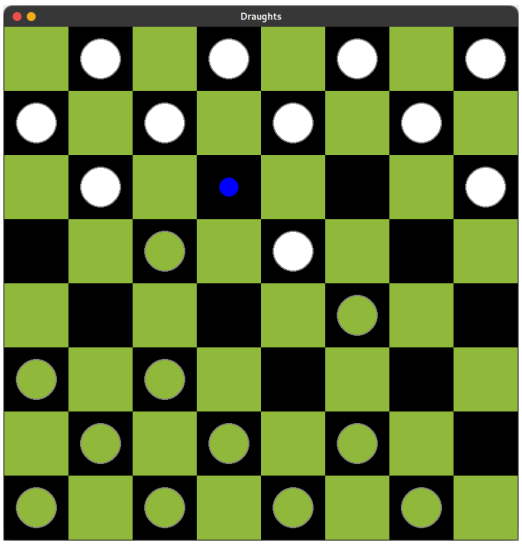
The game uses a click to select & click to place method of movement. So, when a board square is clicked, the x and y co-ordinates are then converted to the boards x and y and then the select method is called with the respective x and y. The select function runs the getValidMoves function to get what moves this piece can make, then stores the piece as class variable so that it can be acessed by the same function when clicked on an empty space. Then to place, the select function checks if the selected square on the board is within the valid moves, if it is then move the piece to the respective place acounting for jumps and intermediate steps

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

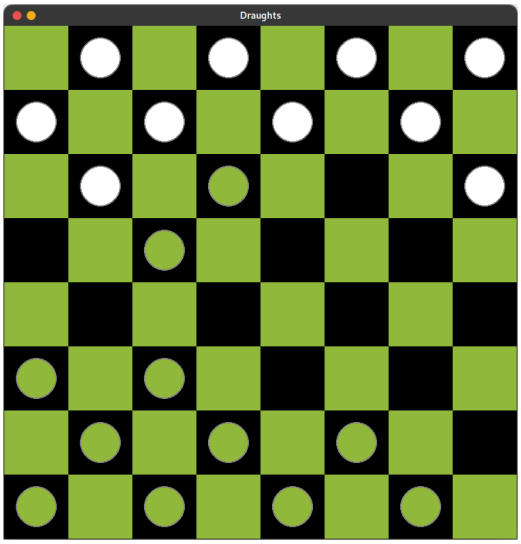
When completing a multi-leg move, the middle step can be chosen and the game waits for the player to make the next move. The move method checks to see if the position is within in the valid moves and is an intermediate step, if it is, then don't change turn otherwise change turn. See figure below 7



(a) Multi-leg before



(b) Multi-leg intermediate



(c) Multi-leg after

Figure 7: Multi-leg movements

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

To access the rules of the game, the help button in the main menu should be clicked. Then the help window pops up, from here you can return back to main menu. See figure below 8

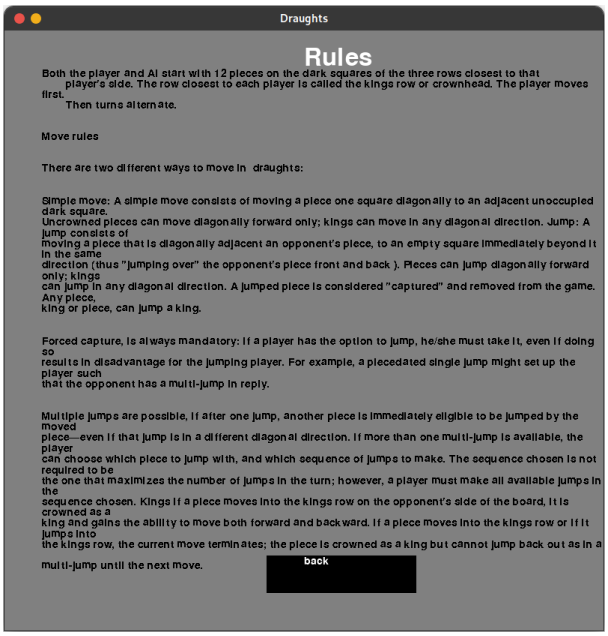


Figure 8: Window showing the rules of the game

References

[1] Python Checkers AI Tutorial.

[2] DaFluffyPotato. Menus - Pygame Tutorial, January 2020.

[3] Sebastian Lague. Algorithms Explained – minimax and alpha-beta pruning, April 2018.

[4] Tech With Tim. Pygame Checkers Tutorial, September 2020.

3 Appendix

3.1 main.py

```
1  import sys
2
3  import pygame
4
5  from draughts.constants import WIDTH, HEIGHT, SQUARE_SIZE, WHITE
6  from draughts.gameManager import GameManager
7  from draughts.minimaxAlgorithm import minimax
8
9  FPS = 60
10 WIN = pygame.display.set_mode((WIDTH, HEIGHT))
11 pygame.display.set_caption("Draughts")
12
13
14 def getRowColFromMouse(pos):
15     x, y = pos
16     row = y // SQUARE_SIZE
17     col = x // SQUARE_SIZE
18     return row, col
19
20
21 def drawText(text, font, color, surface, x, y):
```

```
22      textobj = font.render(text, 1, color)
23      textrect = textobj.get_rect()
24      textrect.topleft = (x, y)
25      surface.blit(textobj, textrect)
26
27
28  def drawMultiLineText(surface, text, pos, font, color=pygame.Color('black')):
29      words = [word.split(' ') for word in text.splitlines()]  # 2D array where
            each row is a list of words.
30      space = font.size(' ')[0]  # The width of a space.
31      max_width, max_height = surface.get_size()
32      x, y = pos
33      for line in words:
34          for word in line:
35              word_surface = font.render(word, 0, color)
36              word_width, word_height = word_surface.get_size()
37              if x + word_width >= max_width:
38                  x = pos[0]  # Reset the x.
39                  y += word_height  # Start on new row.
40              surface.blit(word_surface, (x, y))
41              x += word_width + space
42          x = pos[0]  # Reset the x.
43          y += word_height  # Start on new row.
44
45
46  def main():
47      pygame.init()
48      screen = pygame.display.set_mode((WIDTH, HEIGHT))
49      menuClock = pygame.time.Clock()
50      click = False
51      width = screen.get_width()
52      font = pygame.font.SysFont(None, 25)
53      difficulty = 0
54
55      while True:
56          # menu
57          screen.fill((128, 128, 128))
58          drawText('Main Menu', font, (255, 255, 255), screen, width / 2, 20)
59
60          mx, my = pygame.mouse.get_pos()
61
62          easy = pygame.Rect(width / 2 - 50, 100, 200, 50)
63          pygame.draw.rect(screen, (0, 255, 0), easy)
64          drawText("easy", font, (255, 255, 255), screen, width / 2, 100)
65          medium = pygame.Rect(width / 2 - 50, 200, 200, 50)
66          pygame.draw.rect(screen, (255, 125, 0), medium)
67          drawText("medium", font, (255, 255, 255), screen, width / 2, 200)
68          hard = pygame.Rect(width / 2 - 50, 300, 200, 50)
69          pygame.draw.rect(screen, (255, 0, 0), hard)
70          drawText("hard", font, (255, 255, 255), screen, width / 2, 300)
71          rules = pygame.Rect(width / 2 - 50, 400, 200, 50)
72          pygame.draw.rect(screen, (0, 0, 255), rules)
73          drawText("rules", font, (255, 255, 255), screen, width / 2, 400)
74          quitGame = pygame.Rect(width / 2 - 50, 500, 200, 50)
75          pygame.draw.rect(screen, (0, 0, 0), quitGame)
76          drawText("quit", font, (255, 255, 255), screen, width / 2, 500)
77
78          if easy.collidepoint((mx, my)):
79              if click:
80                  difficulty = 1
81                  break
82          if medium.collidepoint((mx, my)):
83              if click:
84                  difficulty = 2
85                  break
86          if hard.collidepoint((mx, my)):
87              if click:
88                  difficulty = 3
89                  break
90          if rules.collidepoint((mx, my)):
91              if click:
92                  rulesGUI()
93                  break
```

```
94            if quitGame.collidepoint((mx, my)):
95                if click:
96                    pygame.quit()
97                    sys.exit()
98            click = False
99            for event in pygame.event.get():
100               if event.type == pygame.QUIT:
101                   pygame.quit()
102                   sys.exit()
103               if event.type == pygame.MOUSEBUTTONDOWN:
104                   if event.button == 1:
105                       click = True
106
107           pygame.display.update()
108           menuClock.tick(60)
109       if difficulty != 0:
110           game(difficulty)
111
112
113   def rulesGUI():
114       screen = pygame.display.set_mode((WIDTH, HEIGHT))
115       menuClock = pygame.time.Clock()
116       click = False
117       width = screen.get_width()
118       titleFont = pygame.font.SysFont(None, 48)
119       font = pygame.font.SysFont(None, 21)
120       while True:
121           screen.fill((128, 128, 128))
122           drawText("Rules", titleFont, (255, 255, 255), screen, width / 2, 20)
123
124           mx, my = pygame.mouse.get_pos()
125           drawMultiLineText(screen, """Both the player and AI start with 12
                  pieces on the dark squares of the three rows closest to that
126           player's side. The row closest to each player is called the kings row
                  or crownhead. The player moves first.
127           Then turns alternate.
128   \n
129   Move rules
130   \n
131   There are two different ways to move in  draughts:
132   \n
133   Simple move: A simple move consists of moving a piece one square diagonally to
          an adjacent unoccupied dark square.
134   Uncrowned pieces can move diagonally forward only; kings can move in any
          diagonal direction. Jump: A jump consists of
135   moving a piece that is diagonally adjacent an opponent's piece, to an empty
          square immediately beyond it in the same
136   direction (thus "jumping over" the opponent's piece front and back ). Pieces
          can jump diagonally forward only; kings
137   can jump in any diagonal direction. A jumped piece is considered "captured" and
           removed from the game. Any piece,
138   king or piece, can jump a king.
139   \n
140   Forced capture, is always mandatory: if a player has the option to jump, he/she
           must take it, even if doing so
141   results in disadvantage for the jumping player. For example, a piecedated
          single jump might set up the player such
142   that the opponent has a multi-jump in reply.
143   \n
144   Multiple jumps are possible, if after one jump, another piece is immediately
          eligible to be jumped by the moved
145    p i e c e even  if that jump is in a different diagonal direction. If more than
          one multi-jump is available, the player
146   can choose which piece to jump with, and which sequence of jumps to make. The
          sequence chosen is not required to be
147   the one that maximizes the number of jumps in the turn; however, a player must
          make all available jumps in the
148   sequence chosen. Kings If a piece moves into the kings row on the opponent's
          side of the board, it is crowned as a
149   king and gains the ability to move both forward and backward. If a piece moves
          into the kings row or if it jumps into
150   the kings row, the current move terminates; the piece is crowned as a king but
          cannot jump back out as in a
```

```
151 multi-jump until the next move.""", (50, 50), font)
152         back = pygame.Rect(width / 2 - 50, 700, 200, 50)
153         pygame.draw.rect(screen, (0, 0, 0), back)
154         drawText("back", font, (255, 255, 255), screen, width / 2, 700)
155 
156         if back.collidepoint((mx, my)):
157             if click:
158                 main()
159                 break
160 
161         for event in pygame.event.get():
162             if event.type == pygame.QUIT:
163                 pygame.quit()
164                 sys.exit()
165             if event.type == pygame.MOUSEBUTTONDOWN:
166                 if event.button == 1:
167                     click = True
168 
169         pygame.display.update()
170         menuClock.tick(60)
171 
172 
173 def game(difficulty):
174     run = True
175     clock = pygame.time.Clock()
176     gameManager = GameManager(WIN)
177 
178     while run:
179         clock.tick(FPS)
180 
181         if gameManager.turn == WHITE:
182             mm = minimax()
183             value, newBoard = mm.AI(gameManager.getBoard(), difficulty, WHITE,
                    gameManager)
184             gameManager.aiMove(newBoard)
185 
186         if gameManager.winner() != None:
187             print(gameManager.winner())
188             run = False
189 
190         for event in pygame.event.get():
191             if event.type == pygame.QUIT:
192                 run = False
193             if event.type == pygame.MOUSEBUTTONDOWN:
194                 pos = pygame.mouse.get_pos()
195                 row, col = getRowColFromMouse(pos)
196                 # if gameManager.turn == GREEN:
197                 gameManager.select(row, col)
198 
199         gameManager.update()
200         pygame.display.update()
201 
202     pygame.quit()
203 
204 
205 main()
```

3.2 piece.py

```
1  import pygame.draw
2 
3  from draughts.constants import SQUARE_SIZE, GREY, CROWN
4 
5 
6  class Piece:
7      def __init__(self, row, col, colour):
8          self.row = row
9          self.col = col
10         self.colour = colour
11         self.king = False
12         self.x = 0
13         self.y = 0
14         self.calcPosition()
```

```
15          self.padding = 20
16          self.border = 2
17
18      def calcPosition(self):
19          self.x = SQUARE_SIZE * self.col + SQUARE_SIZE // 2
20          self.y = SQUARE_SIZE * self.row + SQUARE_SIZE // 2
21
22      def makeKing(self):
23          self.king = True
24
25      def draw(self, win):
26          radius = SQUARE_SIZE // 2 - self.padding
27          pygame.draw.circle(win, GREY, (self.x, self.y), radius + self.border)
28          pygame.draw.circle(win, self.colour, (self.x, self.y), radius)
29          if self.king:
30              win.blit(CROWN, (self.x - CROWN.get_width() // 2, self.y - CROWN.
                    get_height() // 2))
31
32      def move(self, row, col):
33          self.row = row
34          self.col = col
35          self.calcPosition()
36
37      def __repr__(self):
38          return str(self.colour)
```

3.3 board.py

```
1   import pygame
2
3   from .constants import BLACK, ROWS, GREEN, SQUARE_SIZE, COLS, WHITE
4   from .piece import Piece
5
6
7   class Board:
8       def __init__(self):
9           self.board = []
10          self.greenLeft = self.whiteLeft = 12
11          self.greenKings = self.whiteKings = 0
12          self.createBoard()
13
14      def drawSquares(self, win):
15          win.fill(BLACK)
16          for row in range(ROWS):
17              for col in range(row % 2, ROWS, 2):
18                  pygame.draw.rect(win, GREEN, (row * SQUARE_SIZE, col *
                        SQUARE_SIZE, SQUARE_SIZE, SQUARE_SIZE))
19
20      def createBoard(self):
21          for row in range(ROWS):
22              self.board.append([])
23              for col in range(COLS):
24                  if col % 2 == ((row + 1) % 2):
25                      if row < 3:
26                          self.board[row].append(Piece(row, col, WHITE))
27                      elif row > 4:
28                          self.board[row].append(Piece(row, col, GREEN))
29                      else:
30                          self.board[row].append(None)
31                  else:
32                      self.board[row].append(None)
33
34      def draw(self, win):
35          self.drawSquares(win)
36          for row in range(ROWS):
37              for col in range(COLS):
38                  piece = self.board[row][col]
39                  if piece is not None:
40                      piece.draw(win)
41
42      def move(self, piece, row, col):
43          self.board[piece.row][piece.col], self.board[row][col] = self.board[row
                ][col], self.board[piece.row][piece.col]
```

```
44          piece.move(row, col)
45
46          if row == ROWS - 1 or row == 0:
47              piece.makeKing()
48          if piece.colour == WHITE:
49              self.whiteKings += 1
50          else:
51              self.greenKings += 1
52
53      def remove(self, skipped):
54          for piece in skipped:
55              self.board[piece.row][piece.col] = None
56              if piece is not None:
57                  if piece.colour == GREEN:
58                      self.greenLeft -= 1
59                  else:
60                      self.whiteLeft -= 1
61
62      def getPiece(self, row, col):
63          return self.board[row][col]
64
65      def winner(self):
66          if self.greenLeft <= 0:
67              return WHITE
68          elif self.whiteLeft <= 0:
69              return GREEN
70
71          return None
72
73      def getValidMoves(self, piece):
74          moves = {}
75          forcedCapture = {}
76          left = piece.col - 1
77          right = piece.col + 1
78          row = piece.row
79          if piece.colour == GREEN or piece.king:
80              moves.update(self._traverseLeft(row - 1, max(row - 3, -1), -1,
                    piece.colour, left))
81              moves.update(self._traverseRight(row - 1, max(row - 3, -1), -1,
                    piece.colour, right))
82          if piece.colour == WHITE or piece.king:
83              moves.update(self._traverseLeft(row + 1, min(row + 3, ROWS), 1,
                    piece.colour, left))
84              moves.update(self._traverseRight(row + 1, min(row + 3, ROWS), 1,
                    piece.colour, right))
85
86          if len(moves.values()) <= 1:
87              return moves
88
89          movesValues = list(moves.values())
90          movesKeys = list(moves.keys())
91
92          forced = {}
93
94          for i in range(len(movesKeys)):
95              if not movesValues[i]:
96                  forced[movesKeys[i]] = moves[movesKeys[i]]
97          if len(forced) != len(moves):
98              forced.clear()
99              for i in range(len(movesKeys)):
100                 if movesValues[i]:
101                     forced[movesKeys[i]] = moves[movesKeys[i]]
102             if len(forced) != len(moves):
103                 for i in range(len(movesKeys)):
104                     if movesValues[i]:
105                         forcedCapture[movesKeys[i]] = moves[movesKeys[i]]
106             else:
107                 forcedCapture = forced
108         else:
109             forcedCapture = forced
110         return forcedCapture
111
112     def scoreOfTheBoard(self):
```

```
113         return self.whiteLeft - self.greenLeft + (self.whiteKings * 0.5 + self.
                greenKings * 0.5)
114
115     def getAllPieces(self, colour):
116         pieces = []
117         for row in self.board:
118             for piece in row:
119                 if piece is not None and piece.colour == colour:
120                     pieces.append(piece)
121         return pieces
122
123     def _traverseLeft(self, start, stop, step, colour, left, skipped=[]):
124         moves = {}
125         last = []
126         for row in range(start, stop, step):
127             if left < 0:
128                 break
129             mvs = self._traverse(row, left, skipped, moves, step, last, colour)
130             if mvs is None:
131                 break
132             elif isinstance(mvs, list):
133                 last = mvs
134             else:
135                 moves.update(mvs)
136             left -= 1
137         return moves
138
139     def _traverseRight(self, start, stop, step, colour, right, skipped=[]):
140         moves = {}
141         last = []
142         for row in range(start, stop, step):
143             if right >= COLS:
144                 break
145
146             mvs = self._traverse(row, right, skipped, moves, step, last, colour
                    )
147             if mvs is None:
148                 break
149             elif isinstance(mvs, list):
150                 last = mvs
151             else:
152                 moves.update(mvs)
153
154             right += 1
155         return moves
156
157     def _traverse(self, row, col, skipped, moves, step, last, colour):
158         current = self.board[row][col]
159         if current is None:
160             if skipped and not last:
161                 return None
162             elif skipped:
163                 moves[(row, col)] = last + skipped
164             else:
165                 moves[(row, col)] = last
166
167             if last:
168                 if step == -1:
169                     rowCalc = max(row - 3, 0)
170                 else:
171                     rowCalc = min(row + 3, ROWS)
172                 moves.update(self._traverseLeft(row + step, rowCalc, step,
                        colour, col - 1, skipped=last))
173                 moves.update(self._traverseRight(row + step, rowCalc, step,
                        colour, col + 1, skipped=last))
174             return None
175         elif current.colour == colour:
176             return None
177         else:
178             last = [current]
179             return last
```

3.4 gameManager.py

```
1  import pygame
2
3  from draughts.board import Board
4  from draughts.constants import GREEN, WHITE, BLUE, SQUARE_SIZE
5
6
7  class GameManager:
8
9      def __init__(self, win):
10         self._init()
11         self.win = win
12
13     def _init(self):
14         self.selected = None
15         self.board = Board()
16         self.turn = GREEN
17         self.validMoves = {}
18         self.legCount = 0
19
20     def update(self):
21         self.board.draw(self.win)
22         self.drawValidMoves(self.validMoves)
23         pygame.display.update()
24
25     def reset(self):
26         self._init()
27
28     def select(self, row, col):
29         if self.selected:
30             result = self._move(row, col)
31             if not result:
32                 self.selected = None
33                 self.select(row, col)
34         piece = self.board.getPiece(row, col)
35         if piece is not None and piece.colour == self.turn:
36             self.selected = piece
37             self.validMoves = self.board.getValidMoves(piece)
38             return True
39
40     def _move(self, row, col):
41         piece = self.board.getPiece(row, col)
42         if self.selected and piece is None and (row, col) in self.validMoves:
43             self.board.move(self.selected, row, col)
44             skipped = self.validMoves[row, col]
45             if self.validMoves[list(self.validMoves.keys())[0]]:
46                 if self.validMoves[list(self.validMoves.keys())[0]][0].king:
47                     self.selected.makeKing()
48             if skipped:
49                 self.board.remove(skipped)
50                 if len(self.validMoves) > 1:
51                     del self.validMoves[list(self.validMoves.keys())[0]]
52                 else:
53                     self.changeTurn()
54             else:
55                 self.changeTurn()
56         else:
57             return False
58         return True
59
60     def changeTurn(self):
61         self.validMoves = {}
62         if self.turn == GREEN:
63             self.turn = WHITE
64         else:
65             self.turn = GREEN
66
67     def drawValidMoves(self, moves):
68         for row, col in moves:
69             pygame.draw.circle(self.win, BLUE,
70                                (col * SQUARE_SIZE + SQUARE_SIZE // 2, row *
                                    SQUARE_SIZE + SQUARE_SIZE // 2), 15)
```

```
71
72      def winner(self):
73          return self.board.winner()
74
75      def getBoard(self):
76          return self.board
77
78      def aiMove(self, board):
79          self.board = board
80          self.changeTurn()
```

3.5 minimaxAlgorithm.py

```
1   from copy import deepcopy
2   from math import inf
3
4   from draughts.constants import GREEN, WHITE
5
6
7   class minimax():
8
9       def AI(self, board, depth, maxPlayer, gameManager):
10          if depth == 0 or board.winner() is not None:
11              return board.scoreOfTheBoard(), board
12
13          if maxPlayer:
14              maxEval = -inf
15              bestMove = None
16              for move in self._getAllMoves(board, WHITE):
17                  evaluation = self.AI(move, depth - 1, False, gameManager)[0]
18                  maxEval = max(maxEval, evaluation)
19                  if maxEval == evaluation:
20                      bestMove = move
21              return maxEval, bestMove
22          else:
23              minEval = inf
24              bestMove = None
25              for move in self._getAllMoves(board, GREEN):
26                  evaluation = self.AI(move, depth - 1, True, gameManager)[0]
27                  minEval = min(minEval, evaluation)
28                  if minEval == evaluation:
29                      bestMove = move
30              return minEval, bestMove
31
32      def _simulateMove(self, piece, move, board, skip):
33          board.move(piece, move[0], move[1])
34          if skip:
35              board.remove(skip)
36
37          return board
38
39      def _getAllMoves(self, board, colour):
40          moves = []
41
42          for piece in board.getAllPieces(colour):
43              validMoves = board.getValidMoves(piece)
44              for move, skip in validMoves.items():
45                  tempBoard = deepcopy(board)
46                  tempPiece = tempBoard.getPiece(piece.row, piece.col)
47                  newBoard = self._simulateMove(tempPiece, move, tempBoard, skip)
48                  moves.append(newBoard)
49          return moves
```

3.6 constants.py

```
1   import pygame
2
3   WIDTH, HEIGHT = 800, 800
4   ROWS, COLS = 8, 8
5   SQUARE_SIZE = WIDTH // COLS
6
7   # RGB color
8
```

```
 9  GREEN = (144, 184, 59)
10  WHITE = (255, 255, 255)
11  BLACK = (0, 0, 0)
12  BLUE = (0, 0, 255)
13  GREY = (128, 128, 128)
14
15  CROWN = pygame.transform.scale(pygame.image.load("./draughts/assets/crown.png")
        , (45, 25))
```