PROJECT TOPIC: AI ACTION SELECTION CHECKERS GAME

Overview:

Checkers is game that is played by everyone irrespective of their age. Since this game is manually made and played at homes. I would be creating this game so that families, friends, or individuals can play it online, without the need of any board or even gathering to play it together. This game will connect people and people will play this game with a computer which will make it even more interesting. An AI concept will be incorporated which will train the computer to make the right decision and make laying checkers fun!

Modules and implementation of the modules:

i. The checkers board designing: Designing an attractive GUI board of checkers that will be displayed on the screen which will include:
   a. The size of the board.
   b. Assigning the coloured pieces to the computer and the user.
   c. On the click of each piece it should move.
   d. Giving 3 ideas in prior for performing the best move.

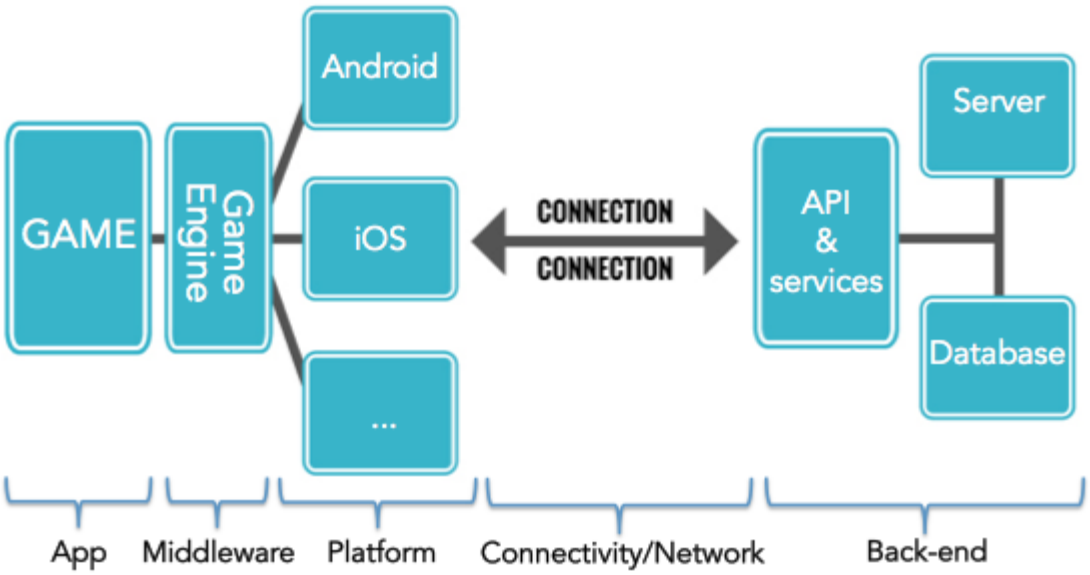
ii. Piece placement: The pieces placement will done accordingly, the user pieces on one side of the board and the computer pieces on the other side of the board.

iii. Main game: this module will be a main programming core, which will have the functionality to change turns, move the pieces, the valid moves, etc..

iv. User interface: the users turn to play the game. Since there will be an opponent called the computer with whom the user will play against. The system will give ideas to perform the next move for helping the user.

v. Computer coding: for the computers turn to play, an AI algorithm known as the ”Minimax” algorithm will be incorporated which will be trained in order for the computer to smartly perform the next move.

Technologies used:



1. Python
2. Pygame

Timeline for project implementation

Work to be done	Start Date	End Date
UI board design	23-02-2021	27-02-2021
Pieces	28-02-2021	05-03-2021
Creating user turn	06-03-2021	10-03-2021
Implementing AI algorithm	11-03-2021	24-03-2021
Training the algorithm	25-03-2021	10-04-2021
Finalizing the modules	11-04-2021	15-04-2021

06-03-2021

Creating a module around the checkers game and then add an API on top of the game so that we can use it with an AI later on

Checkers>>__init__.py :-Initialize the checkers folder as a package

Main.py

- Setup a pygame display: drawing everything onto
- setup a basic event loop: this will check if we press the mouse, if we press a certain key
- for event in pygame.event.get(): this will essentially check to see if any events have happened at the current time, if yes they will be in this list of events.get

GETTING THE CHECKERS BOARD TO DISPLAY

Main.py

```
import pygame
from checkers.constants import WIDTH, HEIGHT

#creating a module around the checkrs game and then add an API on top of the game
# so that we can use it with an AI later on

#1 setup a pygame display where we will be drawing everything onto.
#2 setup a basic event loop: this will check if we press the mouse, if we press a certain key. MAIN LOOP
#3 setup basic drawings, draw the board for the chessboard, draw the pieces

FPS = 60     #G=Frame per second

WIN = pygame.display.set_mode((WIDTH, HEIGHT))
#WIN: a constant value. WIDTH,HEIGHT: 2 Variables we will define(in constants.py)

#set a caption
pygame.display.set_caption('Checkers Game using Minimax Agent')

#define a main funcion
def main():
    #create an event loop: this will run 'x' times per second which will check everything
    run = True

    #define a clock:define our game to run at a constant frame rate.
    #the clock will make sure that the main event loop wont run too fast or too slow.
    clock = pygame.time.Clock()

    while run:
        clock.tick(FPS)

        #setup the basic event loop for pygame
        for event in pygame.event.get():
            if event.type == pygame.QUIT:    #this means we hit the red button on the top of the screen
                run = False

            if event.type ==  pygame.MOUSEBUTTONDOWN:     #This means we pressed any mouse on our mouse
```

```
down
            pass


   pygame.quit()

main()
```

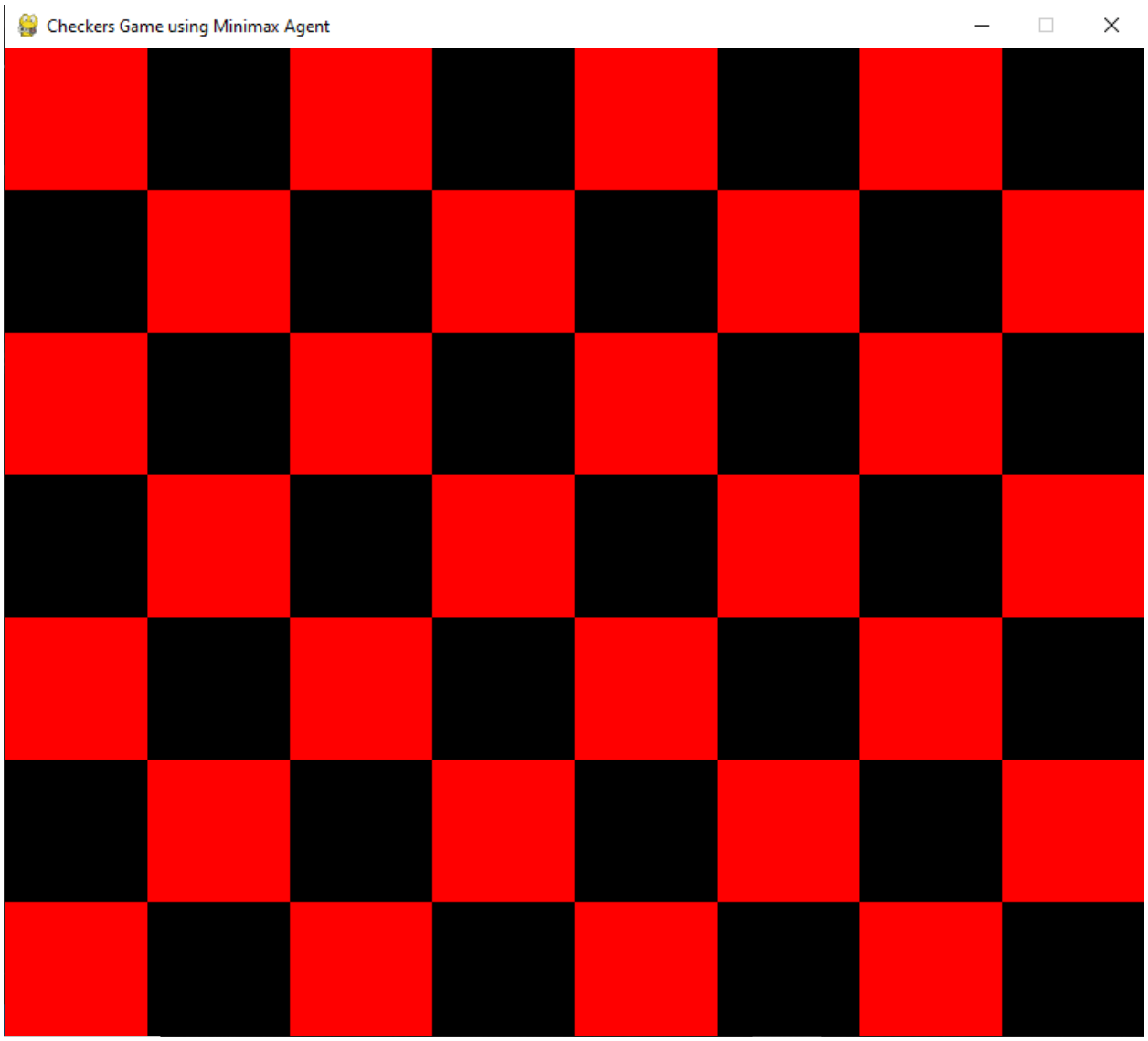
Constants.py

```
#Put all the constant values inside of this file

import pygame

#DEFINE THE HEIGHT AND WIDTH
WIDTH, HEIGHT = 800, 700   #800 pixels

#DEFINE ROWS AND COLS IN CHECKERS BOARD
ROWS, COLS = 8, 8

#HOW BIG IS ONE SQAURE OF THE CHECKER BOARD
SQUARE_SIZE = WIDTH//COLS

#DEFINE VARIABLE FOR COLOURS
#RGB COLOR
RED = (255, 0, 0)
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
BLUE = (0,0, 255)
```

DRAW THE CHECKER BOARD ON THE DISPLAY

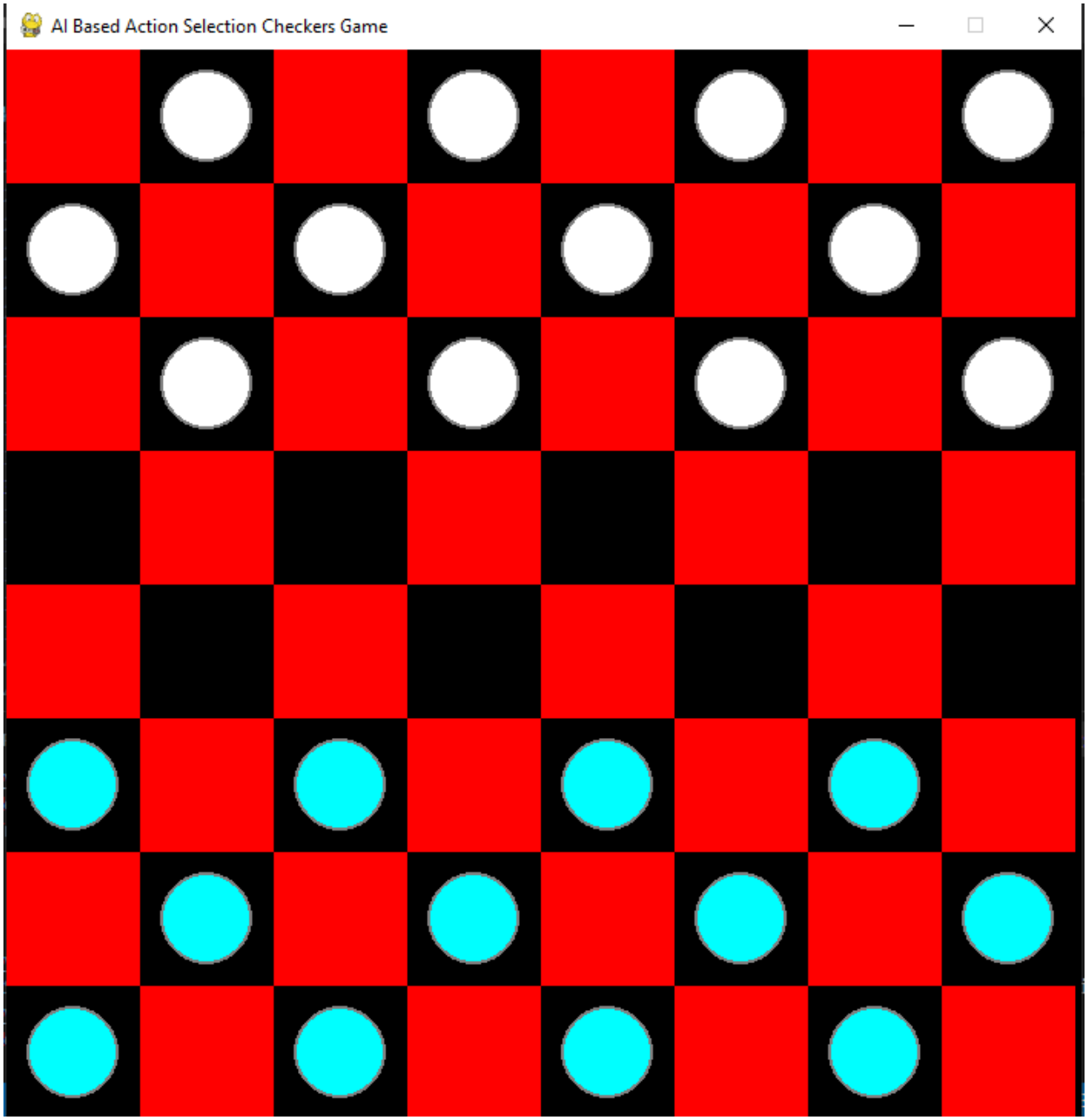
Board.py

```
#Put a class named "Board": This class will represent a checkers board.
#This class will handle all the different pieces moving, leading specific pieves, deleting specific pieces, rawing itself
on the screen.

import pygame
from .constants import BLACK, ROWS, RED, SQUARE_SIZE


class Board:
    def __init__(self):

        #Attributes

        #1. Internal representation of the board
        self.board = []   #creating a 2D list
        #Whose turn is it
        self.selected_piece = None  #have we selected a piece yet, or not
        self.red_left = self.white_left = 12 #keeps track of how many red, how many white pieces we have.
        self.red_kings = self.white_kings = 0

    #A surface/window to draw the red and black cubes on in a checkerboard pattern
    def draw_squares(self, win):
        win.fill(BLACK)
        for row in range(ROWS):
            for col in range(row % 2, ROWS, 2):
                pygame.draw.rect(win, RED,(row*SQUARE_SIZE, col*SQUARE_SIZE, SQUARE_SIZE,
SQUARE_SIZE
                )) #Drawing the red rectangle starting from the top left
```


Checkers Game using Minimax Agent

13-03-2021

PIECES:



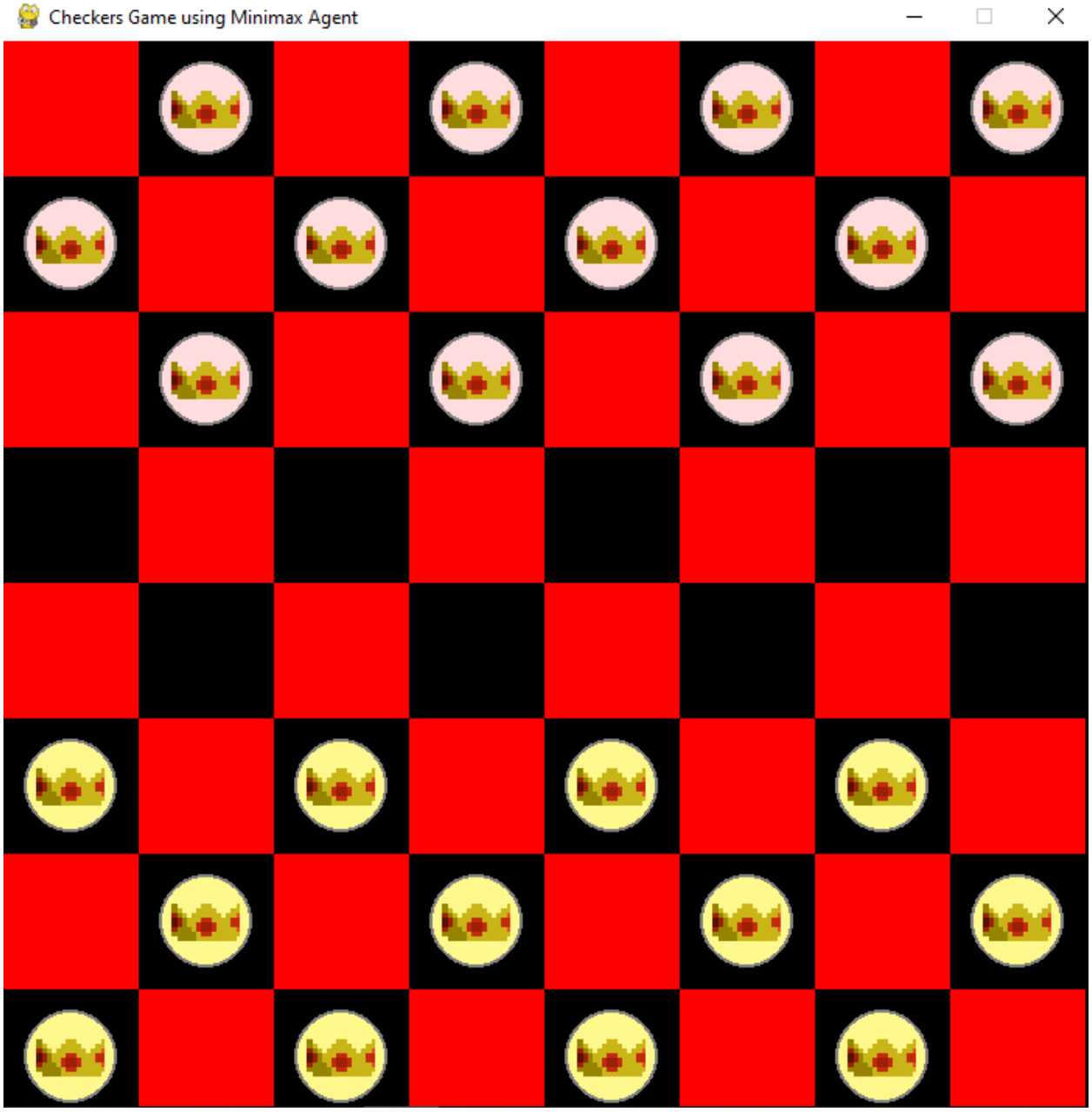
I have successfully added the pieces on the top and the bottom of the board.

MODULE 1 COMPLETED

MODULE 2: **21-03-2021**

Logic of the game, moving of the pieces, valid moves of each piece, and also what happens if one of the pieces is a king.

1. Draw a crown for a king piece.
   In the assets folder, we have a king picture(crown). So whenever the piece is a king piece this image will be assigned to that piece to show that its now a king piece.

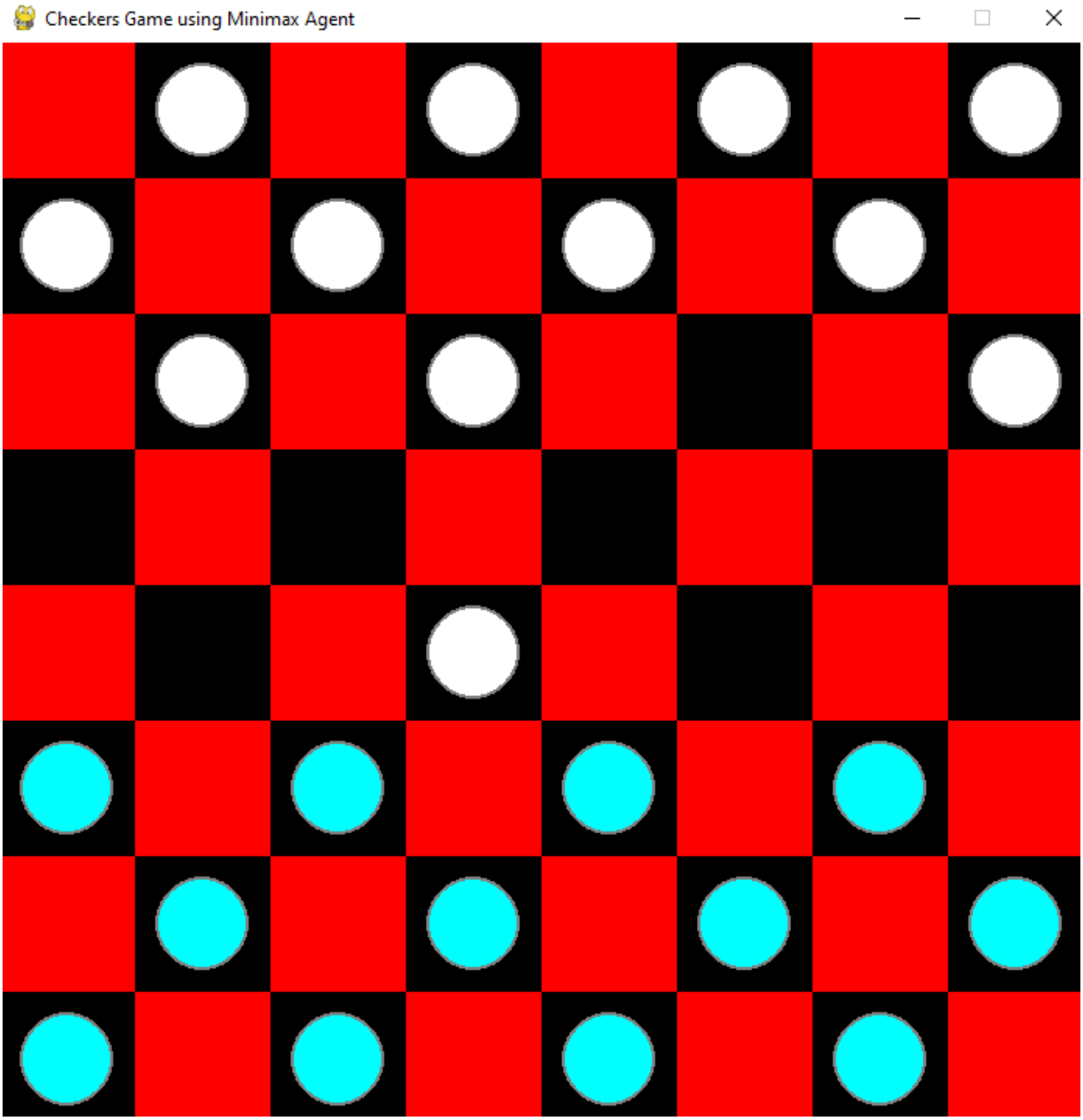


This is just to show how the ing image will look like on the piece.

2. To move pieces around and delete pieces

I have created a method in board.py, names move_pieces which will have the logic to move the pieces.

When we hit the last row or the first row, then the pieces should become the king. We should check that the row col the piece is in then we have to make that iece a king.



The white piece is successfully moving.

MODULE 2 COMPLETE

Module 3: 27-03-2021

1. Jumping
2. Multiple jumping
3. King movement

Make another file inside checkers, call this game.py: this file is responsible for actually handling the game. Whose turn is it?, did we select the piece? Can the piece move here or there?

This file will use board and many other pygame methods

I created a Game class: this class will allow us to interface with the board, the pieces in which i used few simple methods.

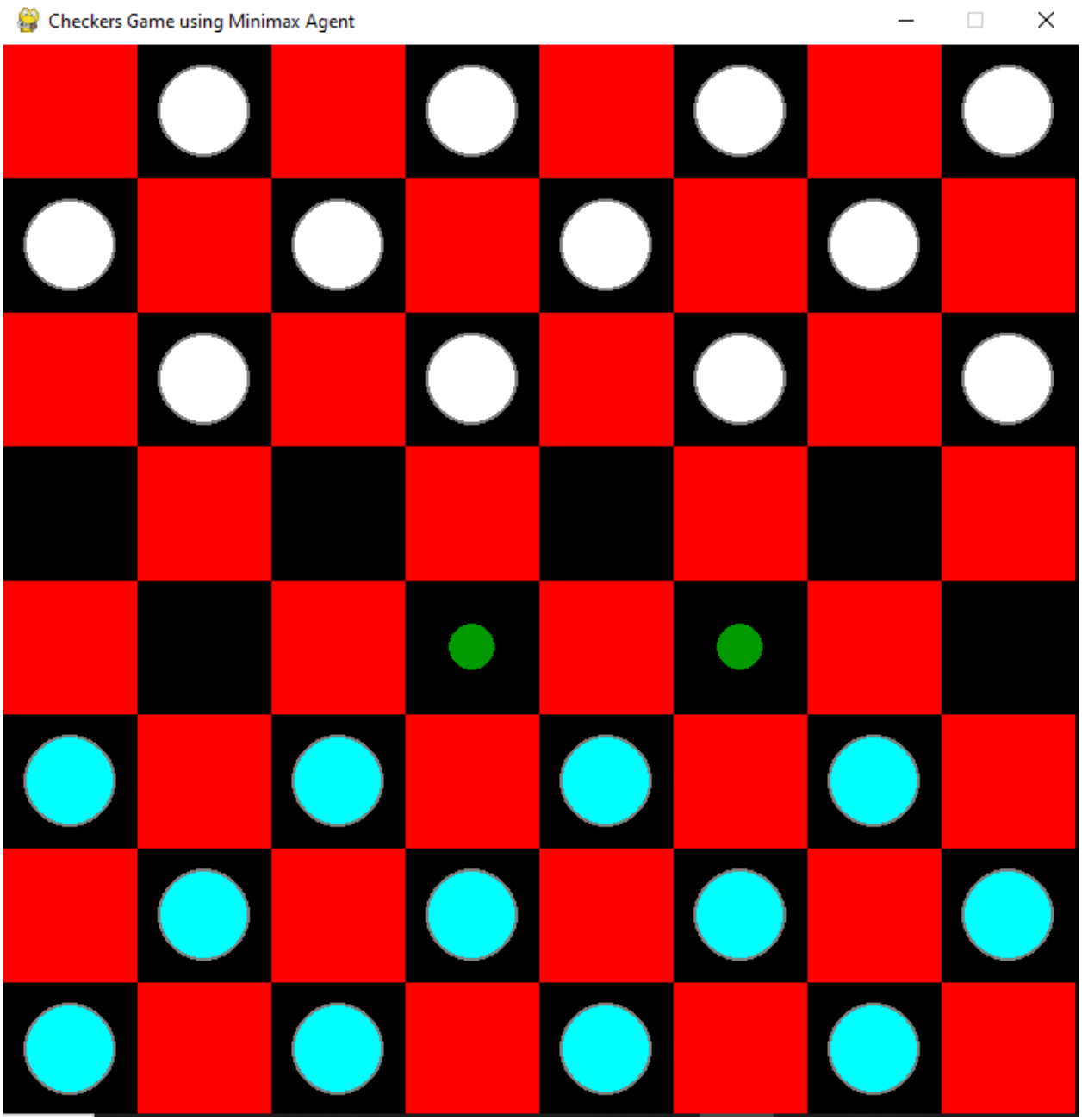
Algorithm for jumping

1. Check color of the piece, that determines the direction we're going to move it in. I have already set the direction set on the piece as Negative or Positive one.
   If RED move down (then check left right or left).
   Checking the left diagonal first, it first checks:
   - Does that place or diagonal have a piece in it? Or
   - Is it blank? Valid move
   - Is the piece BLUE or RED? Is the piece as the same color as our piece then we cant move on that diagonal.
   - If it has an opposite piece color in it, then we have to keep moving diagonally and see if we can jump over that piece and see the other next square below that
   - Same for right diagonal

   DOUBLE JUMPING:

   - Moving 2 pieces ahead

Since Module 3 is very complicated and has a very complex algorithm for jumping and double jumping the piece, I have progressed with half of the coding part. And the rest of module 3 will be completed within this week.



For the valid moves hint: i have defined 2 methods in board.py named:

```
def _traverse_left(self, start, stop, step, color, left, skipped=[]):
        moves = {}
        last = []
        for r in range(start, stop, step):
            if left < 0:
                break   #when we are at the edge of the board on the left side then no further cols we
have

            current = self.board[r][left]
            if current == 0:
```

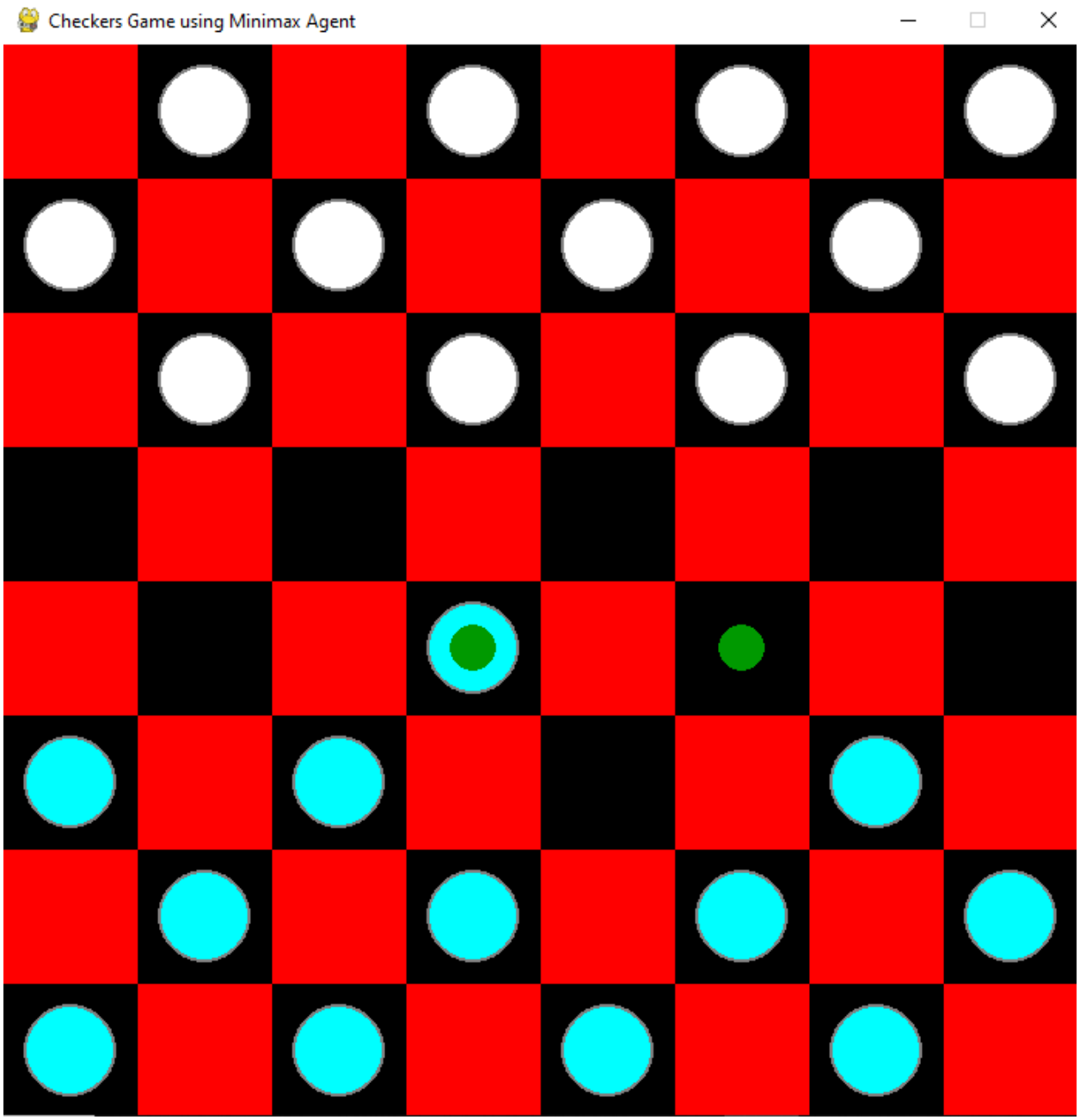
```
            if skipped and not last:
                break
            elif skipped:
                moves[(r, left)] = last + skipped

            else:
                moves[(r, left)] = last

            if last:
                if step == -1:
                    row = max(r-3, 0)
                else:
                    row = min(r+3, ROWS)

                moves.update(self._traverse_left(r+step, row, step, color, left-1, skipped=last))
                moves.update(self._traverse_right(r+step, row, step, color, left+1, skipped=last))
            break

        elif current.color == color:
            break
        else:
            last = [current]

        left -= 1

    return moves
```

SAME FOR RIGHT DIAGONALS

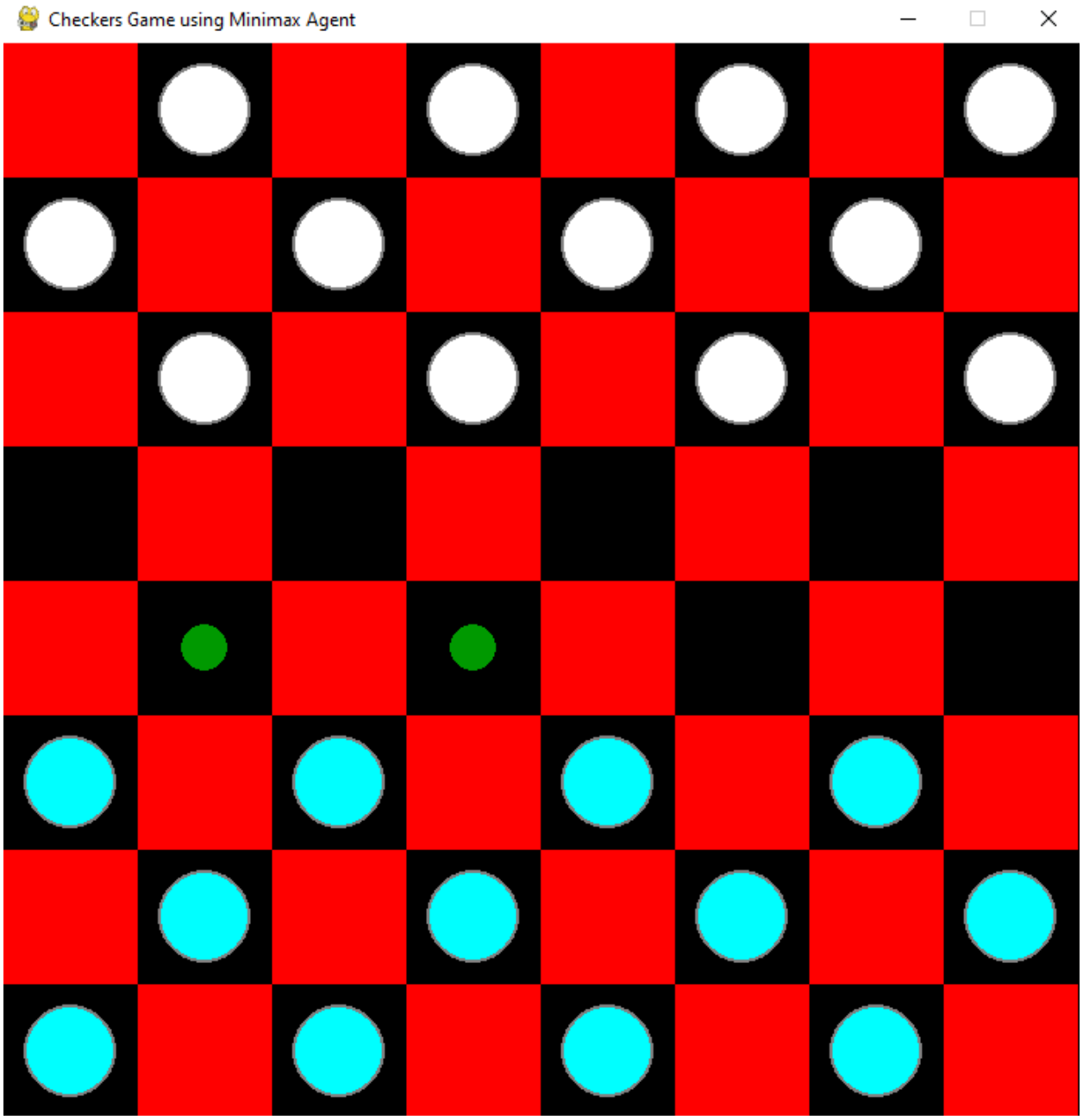
This algorithm checks if the user wants to jump or double jump to another place, it will first check the square, is it empty or not. If empty and its a valid move then it will show or give a hint or a dot(green) pointing at that location.

If there is a piece already present on that square place then it will check:

1. Is the piece the opponents piece?
2. What is the color of the piece?
3. If that piece is of same color as mine, then it wont do the attack or show any hint there.
4. If the piece is of opponents color, and a valid attack move then it will jump to the ahead square in which it can attack the opponent and win.

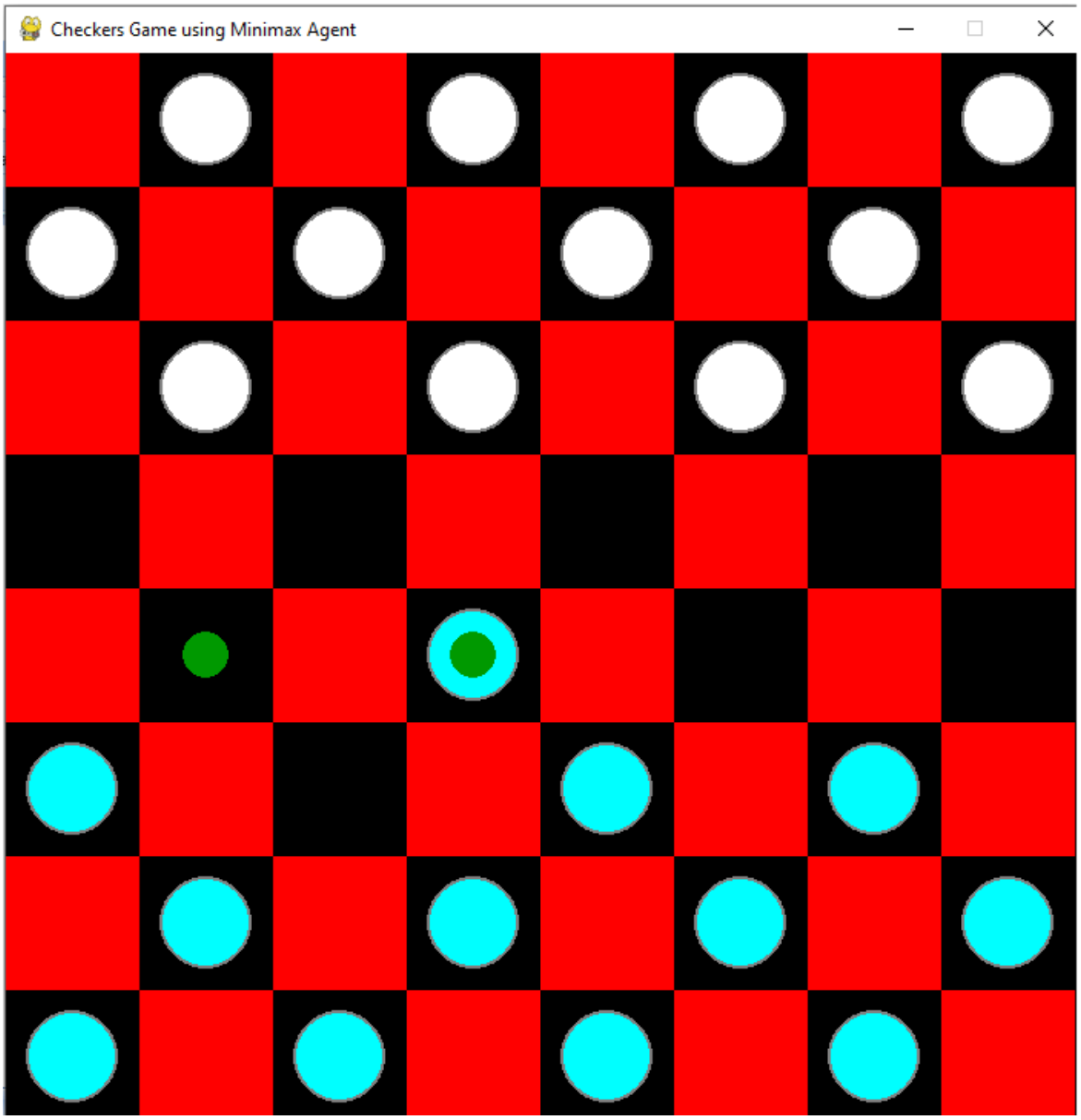


This is now updating the valid pieces and updating the valid moves

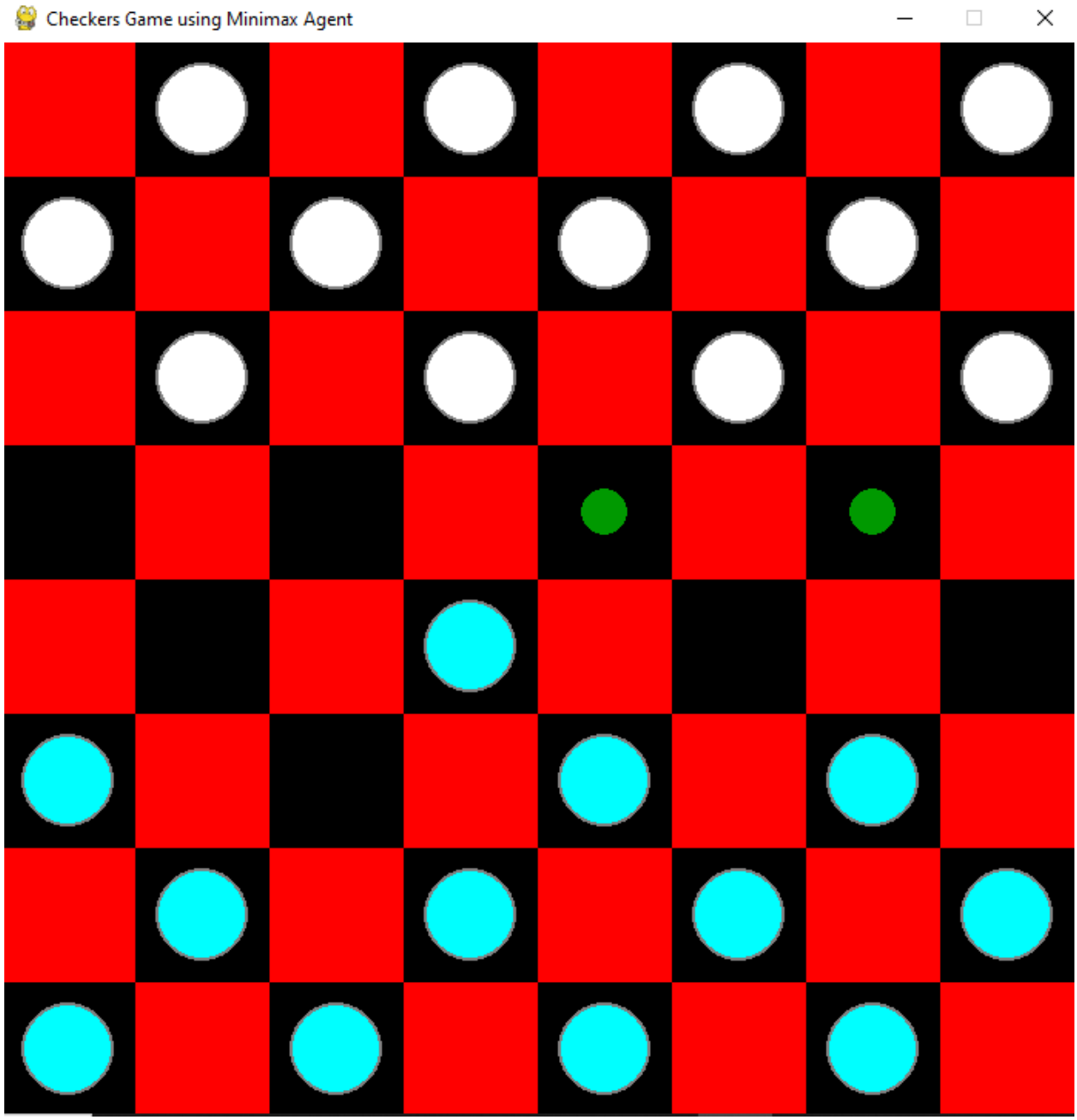


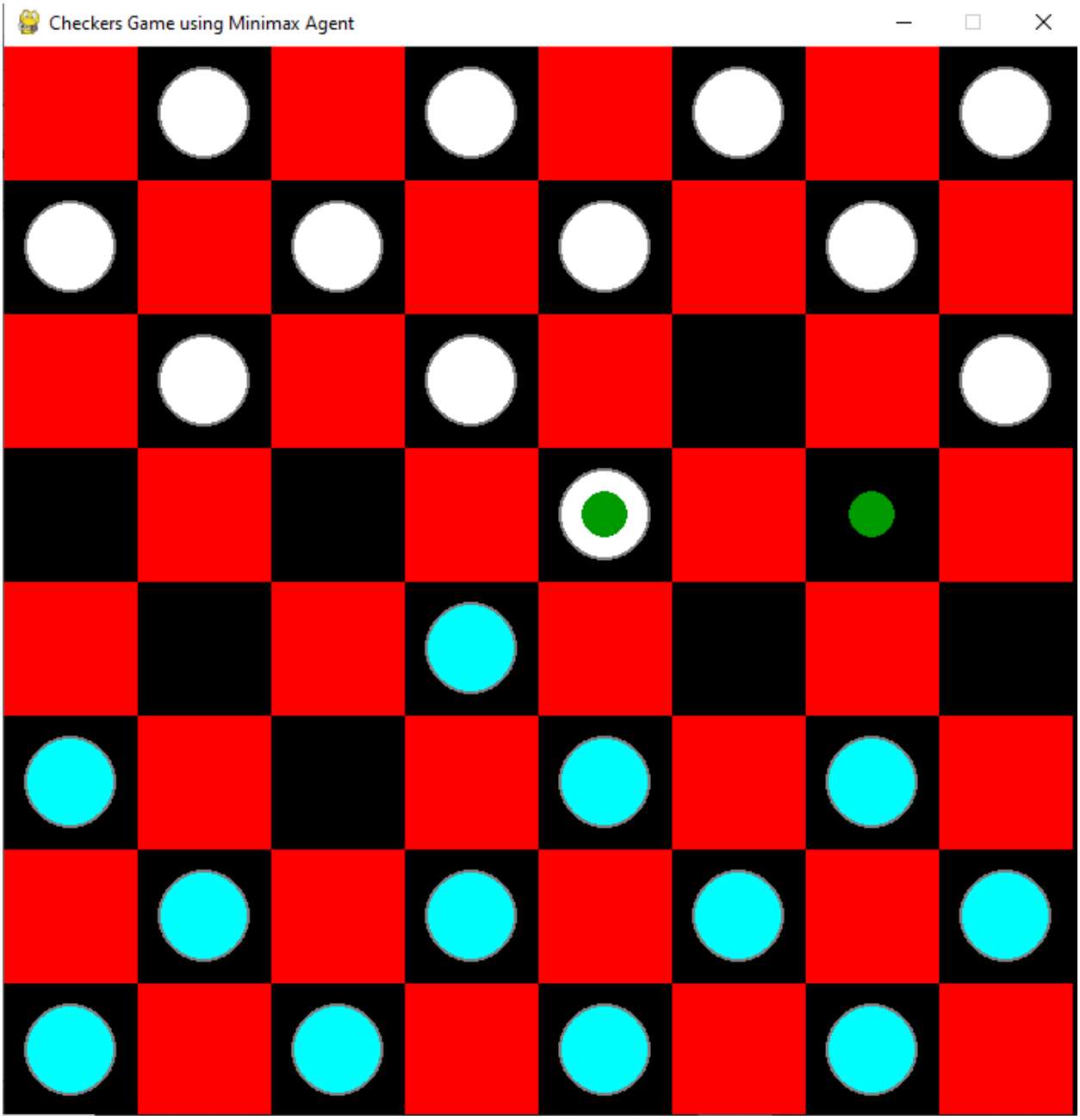
Now this will check the turns:

1. User playes(BLUE)



2. Computer plays(WHITE)





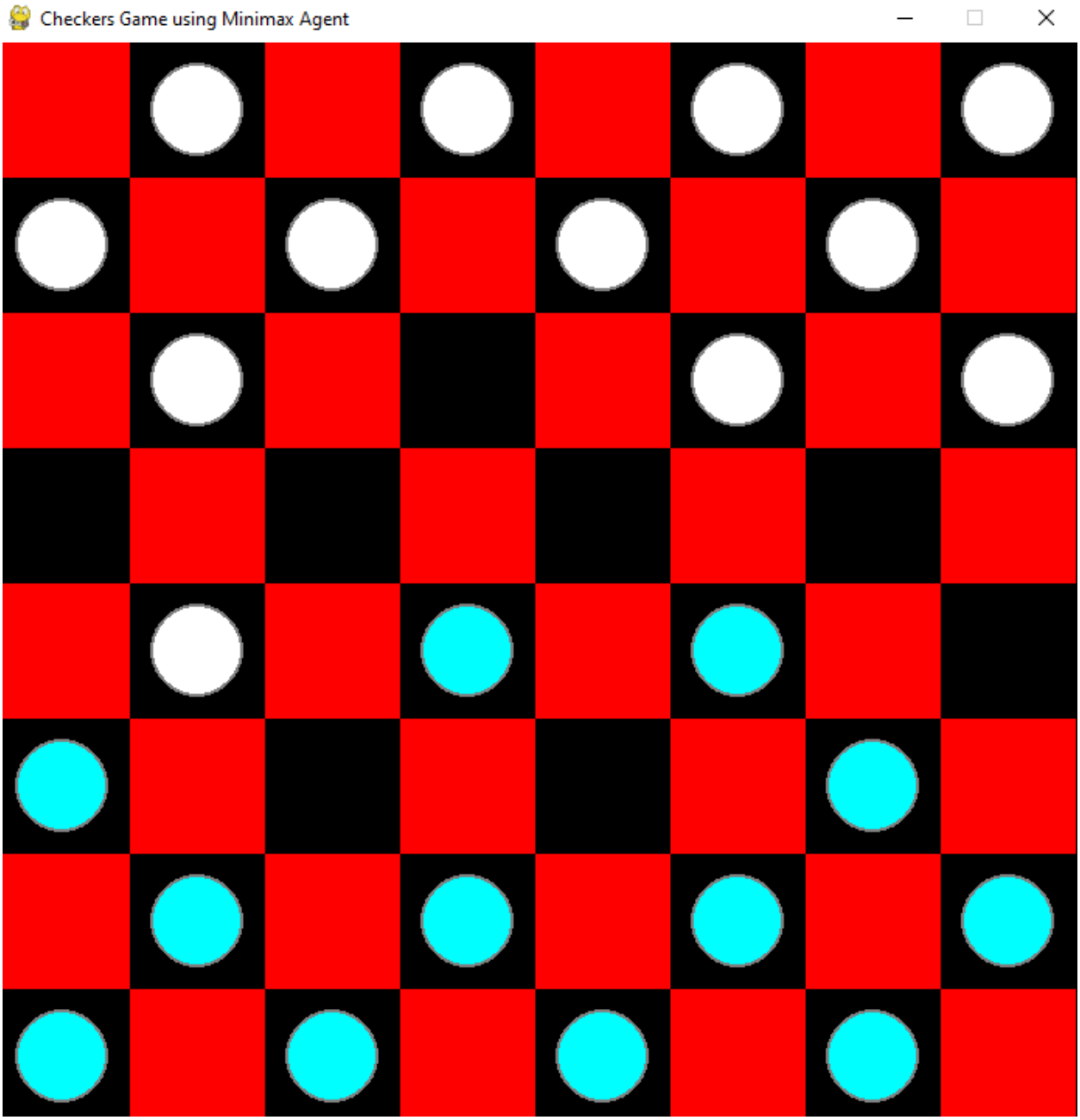
When the user has played its turn, the system wont allow the computer to play and vice versa.

But when playing a move, it doesn't remove the valid hint dots, sp for that:

def change_turn(self):

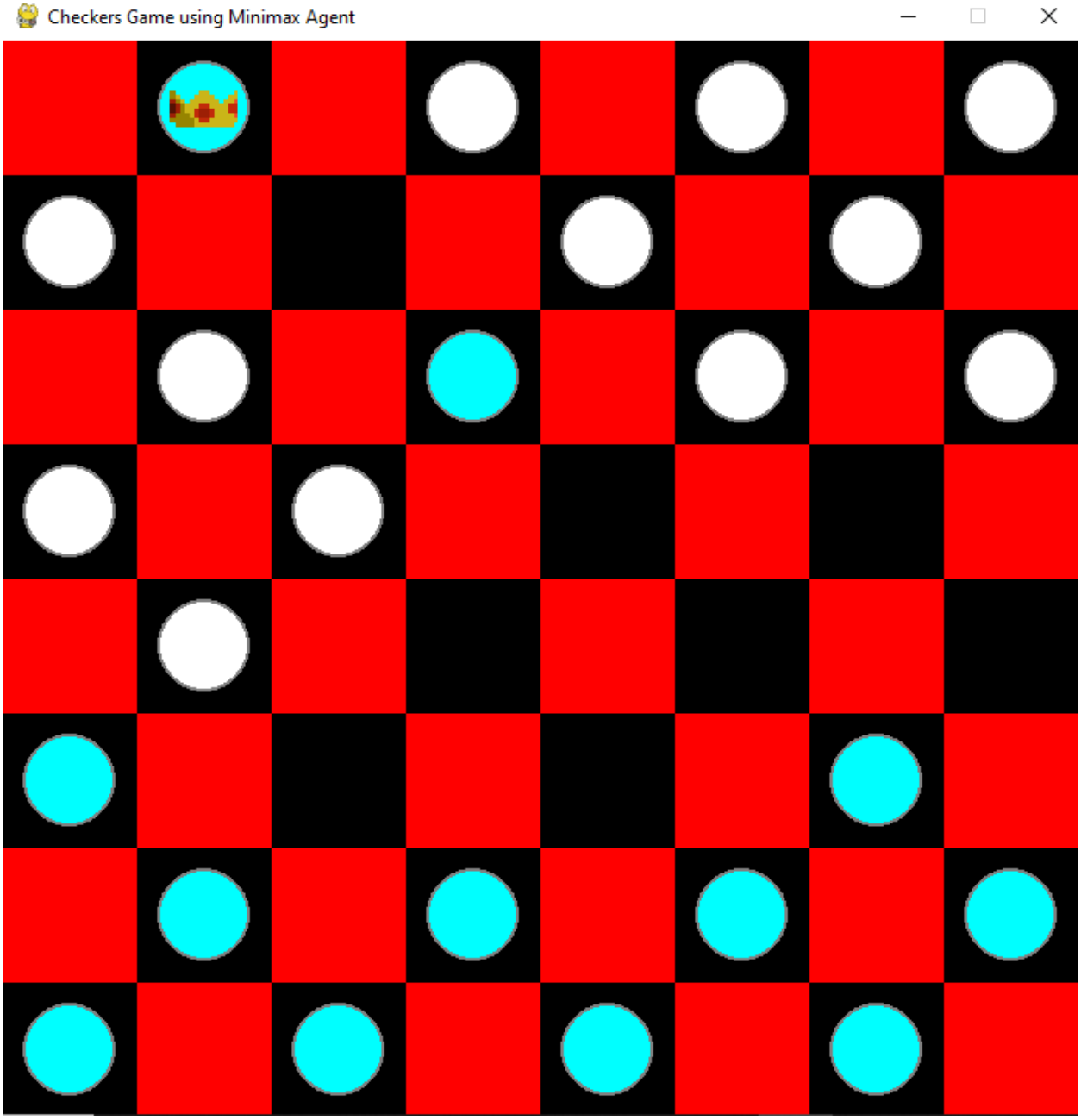
    self.valid_moves = {}

when i made this change in game.py. this will remove the hints after playing a move

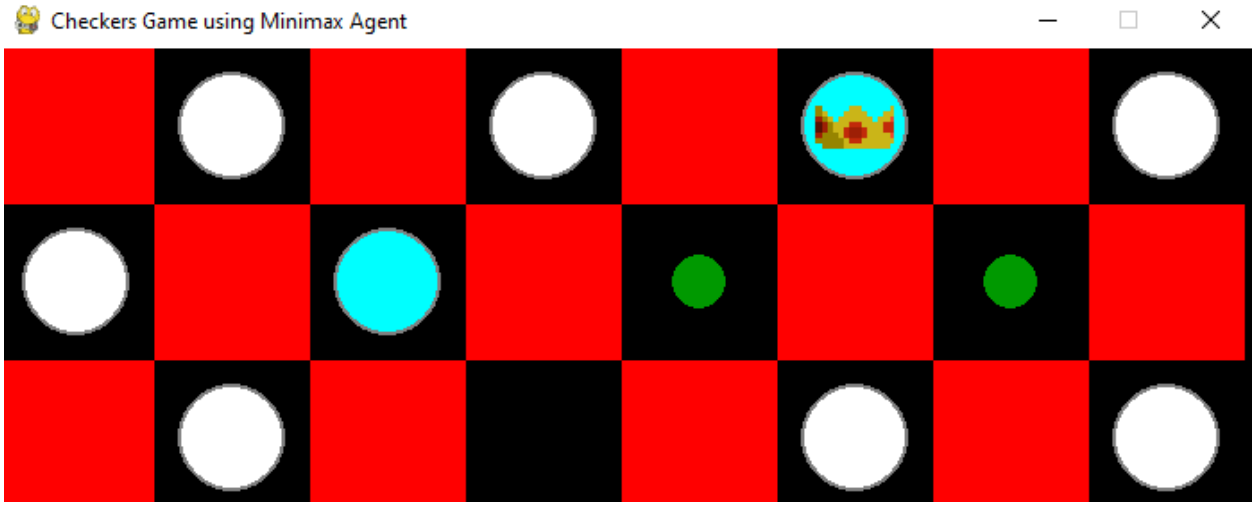


Now the green hints are not appearing

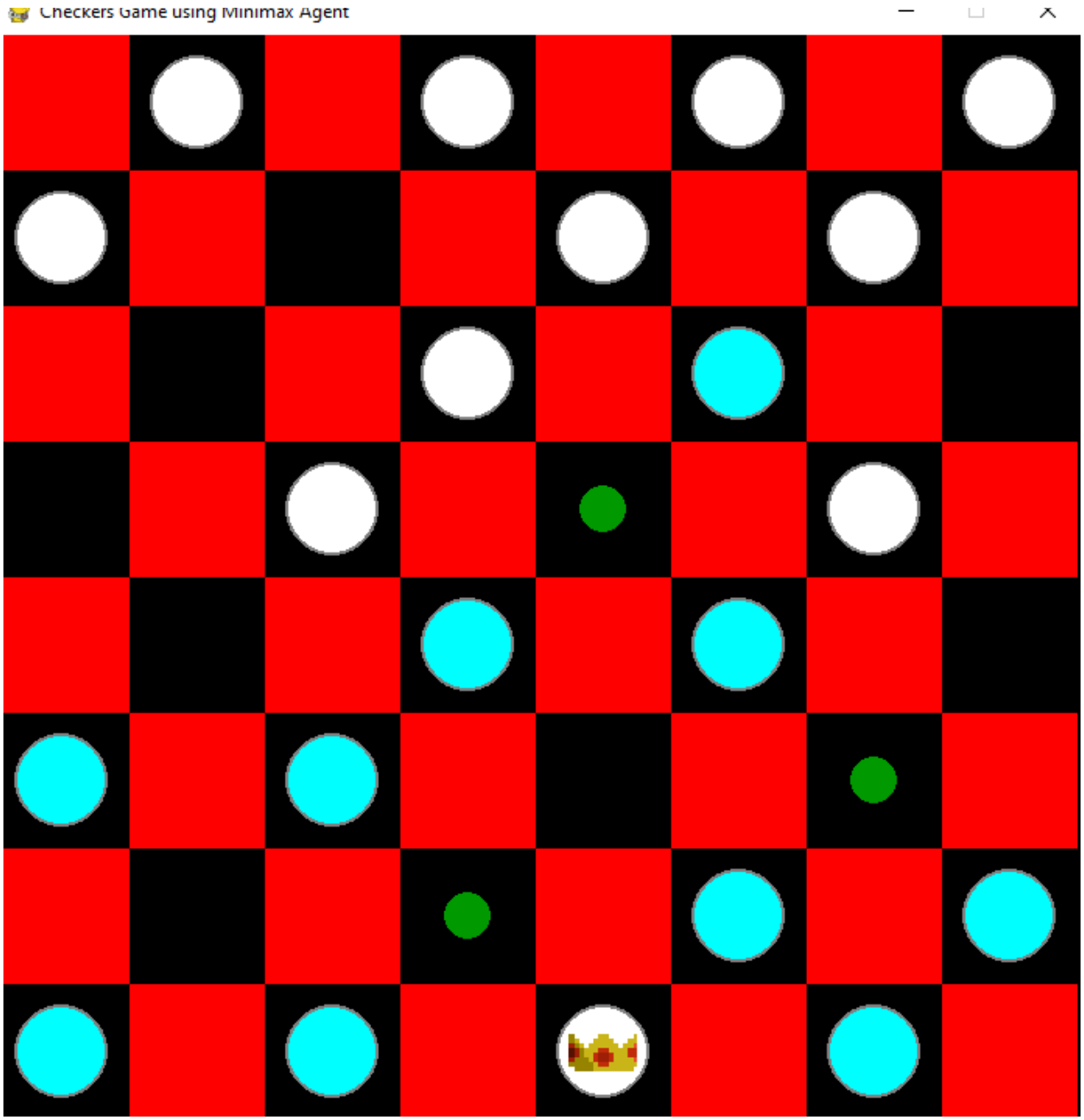
Now checking that if someone has a King piece:



The blue piece has successfully got a King assigned on top of it showing that its now a king piece, because it has read the first row or the 2nd column of the board

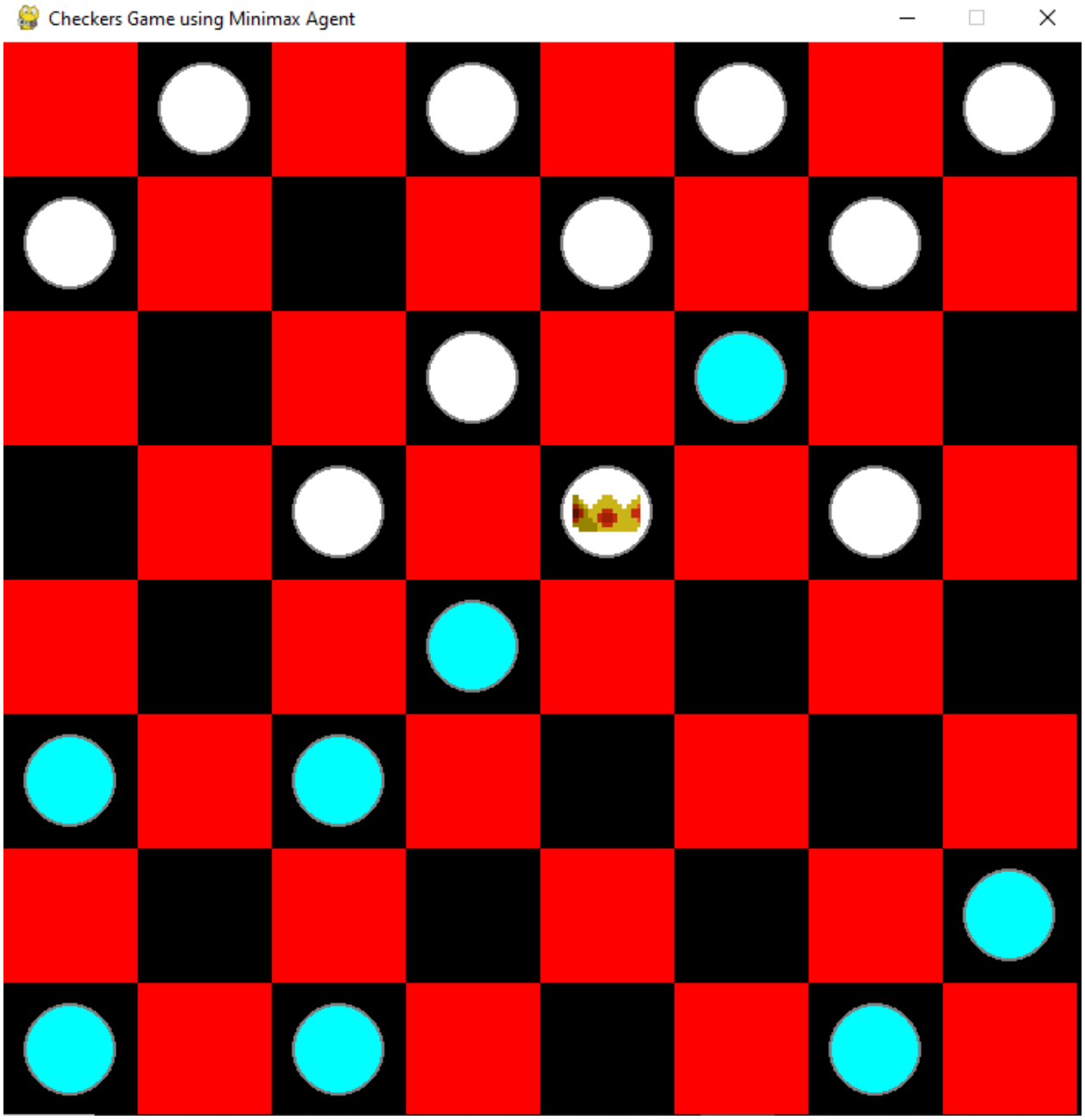


The king piece can also jump backward



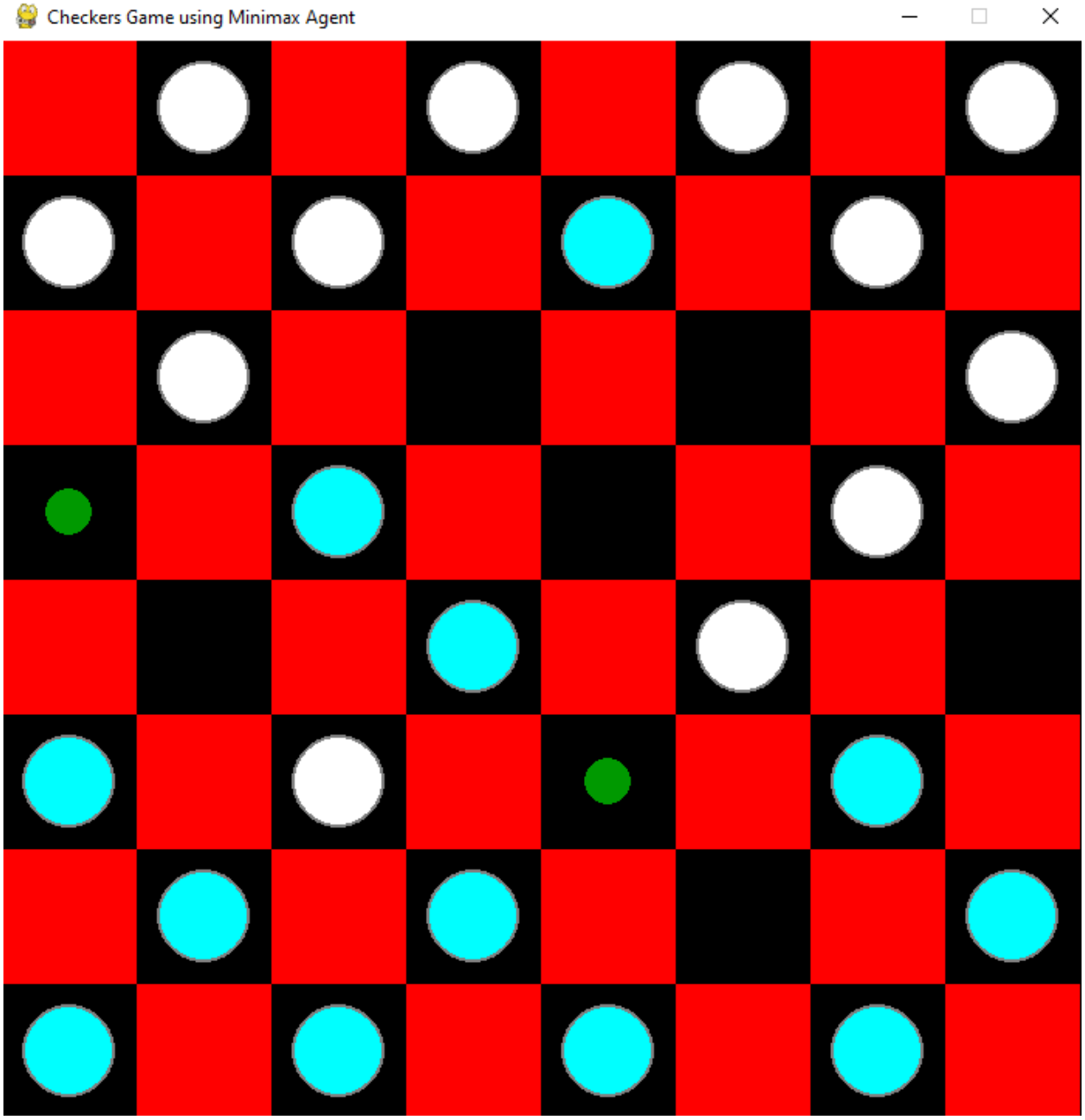
When the king moves backward, the valid hints in green can be because it's a KING.

When the king moves on the top greenest hint

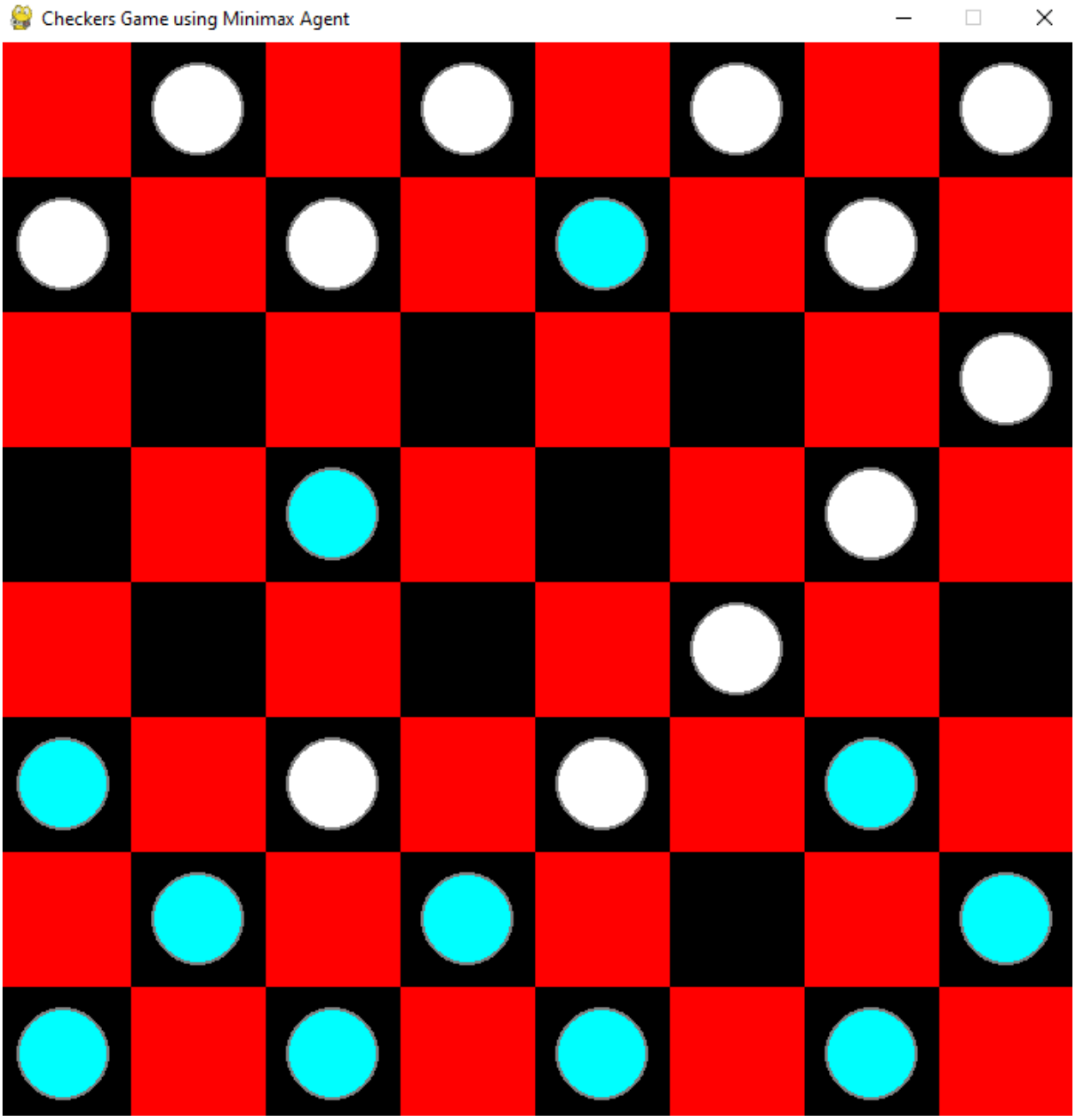


It can easily eat or attack all the opponents' pieces

DOUBLE JUMP:



The white piece can also make a double jump


Checkers Game using Minimax Agent

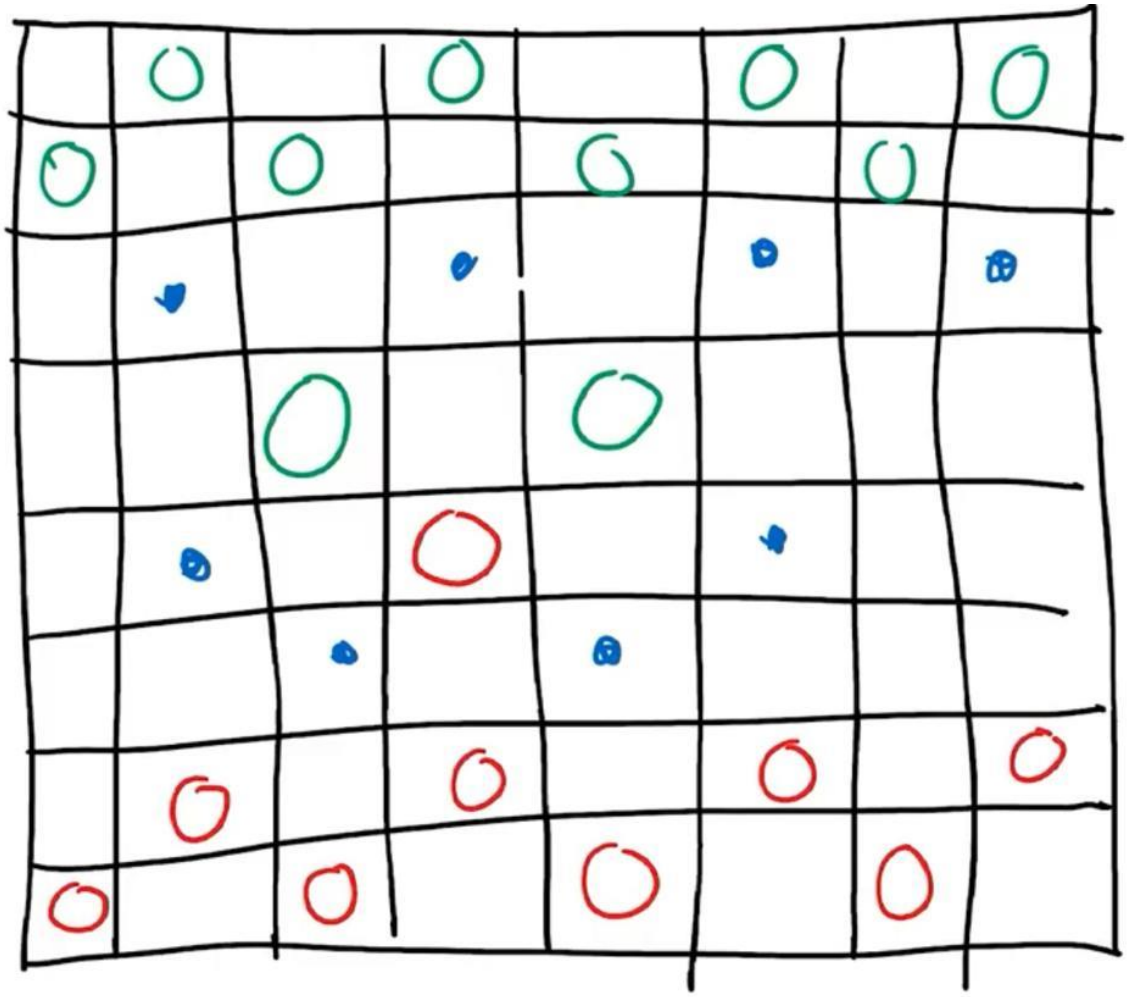
04-04-2021

MODULE 4

MINIMAX ALGORITHM:

Minimax algorithm: checkers is a game that has a lot of different potential moves not as many as chess or any other games. There is a high degree of complexity.

Lets consider the amount of possible moves that this AI Agent can make in its current position



Assuming the green is AI and the red pieces are the Human player.

The blue dots are the valid moves. The blue dots are the 8 potential moves.

The way the Minimax algorithm works is it assuming that our player that person we're playing against is making the best possible moves that it can, so actually we will consider every single possible move that we can potentially make and then for each one of those moves we are going to consider every single move the other player that can potentially make against us.

Essentially we will do is that we want to either maximize or minimises score and the other player wants to maximize or minimize the reverse of what we are doing.

SCORE:

In every single position of this board, I will give it a score. The score will be defined by something that we come up with but a really easy way to determine is just to simply count the number of green pieces and then the number of red pieces. So, we can say that the score is going to be equal to the number of green minus and the number of red pieces.

SCORE = NO. OF GREEN – NO. OF RED.

Therefore green wants to make this as big as possible, it wants to maximize the score and wants to maximize the pieces and red wants to minimize that score. Thus we will use Minimax accordingly.

Therefore, when we make our move we will maximize, pick the move which has the best potential score and red will pick the move which has the minimum potential score.

Algorithm: green want to maximize our score and red wants to minimize its score.

MINIMAX ALG:

- DECISION TREE: red tried to minimize its score and green tries to maximize its score and making the assumption on this that red is going to take the best possible move that it can make from any position.

07-04-2021

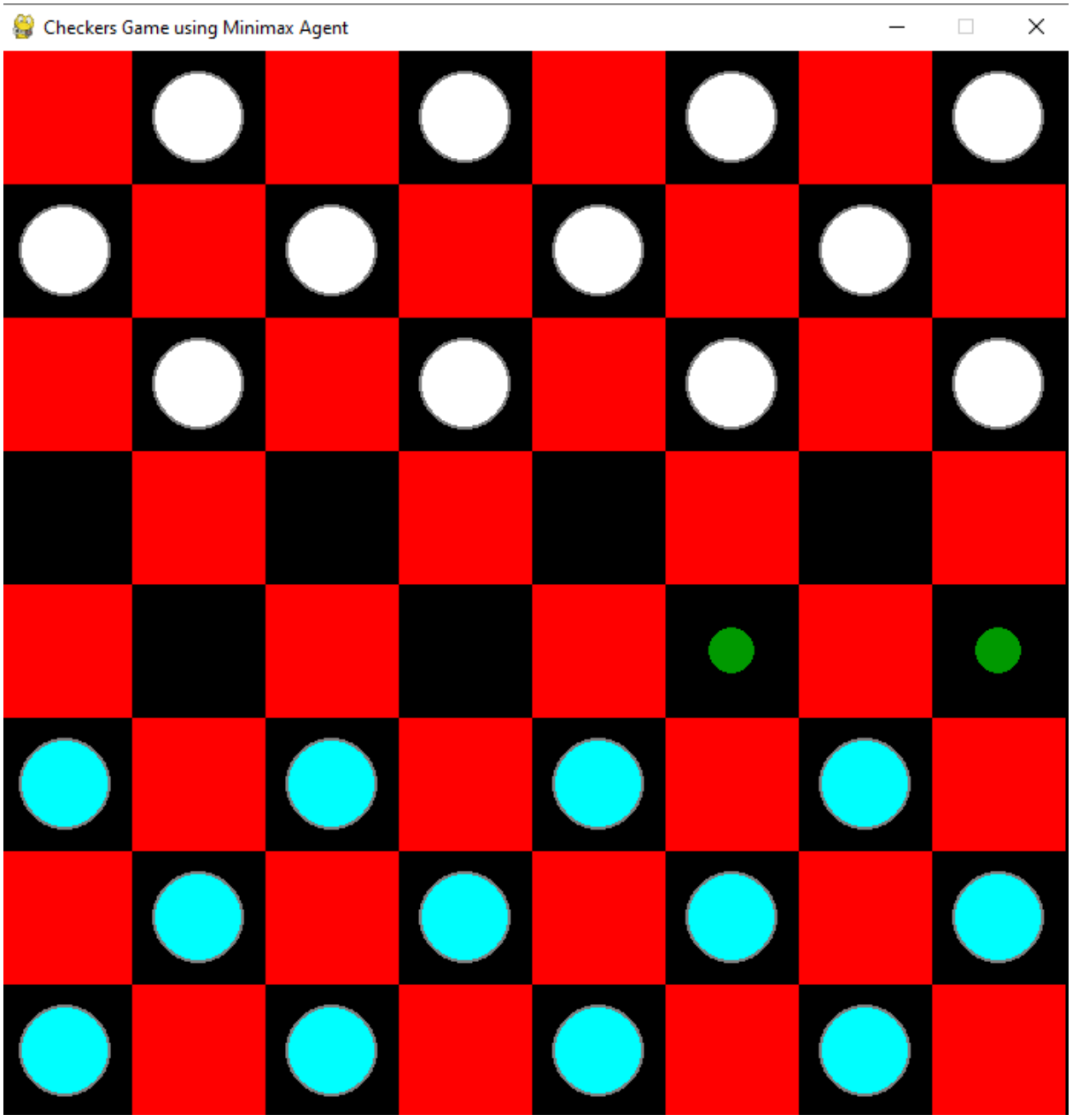
MINIMAX ALG: For implementing this algorithm i have created a folder named “Minimax” and a file named “algorithm” thus when the Minimax is called, it will check the max value which is the Human player in this case and the man function which is the RED pieces the AI player.

Whenever the Human player plays its turn the minimax algorithm is run and the AI player plays its turn automatically.

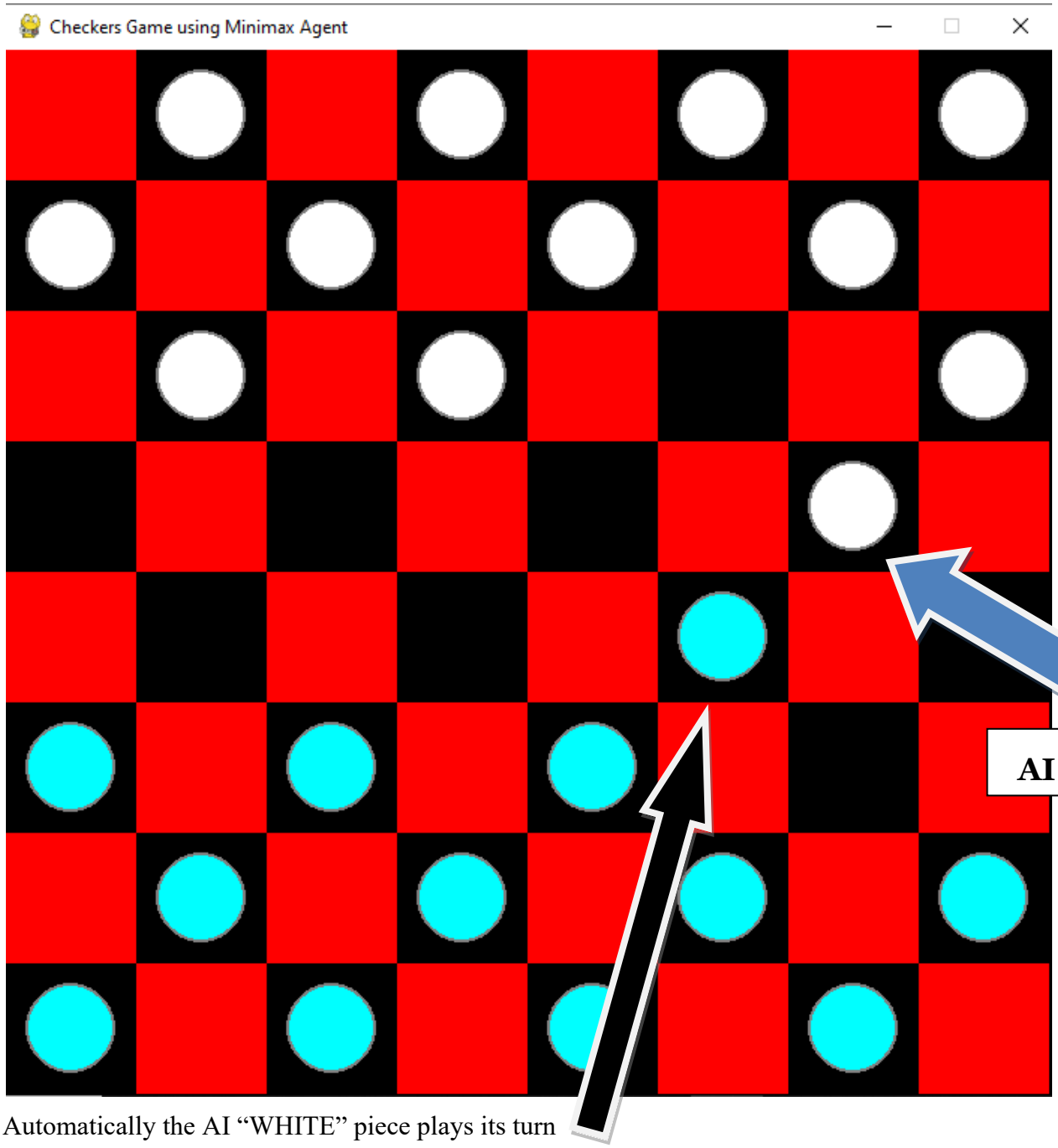
Whenever any of the player wins the game, it returns the color and quits

```
from copy import deepcopy
import pygame

BLUE = (0,255,255)
WHITE = (255, 255, 255)
RED= (255, 0, 0)


#position:- This stands for the current posisiton that we are in.
   #depth:- Tells us how how far or extendeting the tree.
     #This will be a recursive call. we will decrement this
   #max_player: A boolean value, which tells are we minimizing the value or min the value.
   #game:- The actual game object which we will get from main.py file and pass that to the alg

def minimax(position, depth, max_player, game):
   if depth == 0 or position.winner() != None:
      return position.evaluate(), position

   if max_player:
      maxEval = float('-inf')
      best_move = None
      for move in get_all_moves(position, WHITE, game):
         evaluation = minimax(move, depth-1, False, game)[0]
         maxEval = max(maxEval, evaluation)
         if maxEval == evaluation:
            best_move = move

      return maxEval, best_move

   else:
      minEval = float('inf')
      best_move = None
      for move in get_all_moves(position, BLUE, game):
         evaluation = minimax(move, depth-1, True, game)[0]
         minEval = min(minEval, evaluation)
         if minEval == evaluation:
            best_move = move
```

```
        return minEval, best_move


def simulate_move(piece, move, board, game, skip):
    board.move_pieces(piece, move[0], move[1])
    if skip:
        board.remove(skip)

    return board

def get_all_moves(board, color, game):
    moves = []

    for piece in board.get_all_pieces(color):
        valid_moves = board.get_valid_moves(piece)
        for move, skip in valid_moves.items():
            temp_board = deepcopy(board)
            temp_piece = temp_board.get_piece(piece.row, piece.col)
            new_board = simulate_move(temp_piece, move, temp_board, game, skip)
            moves.append(new_board)
    return moves
```

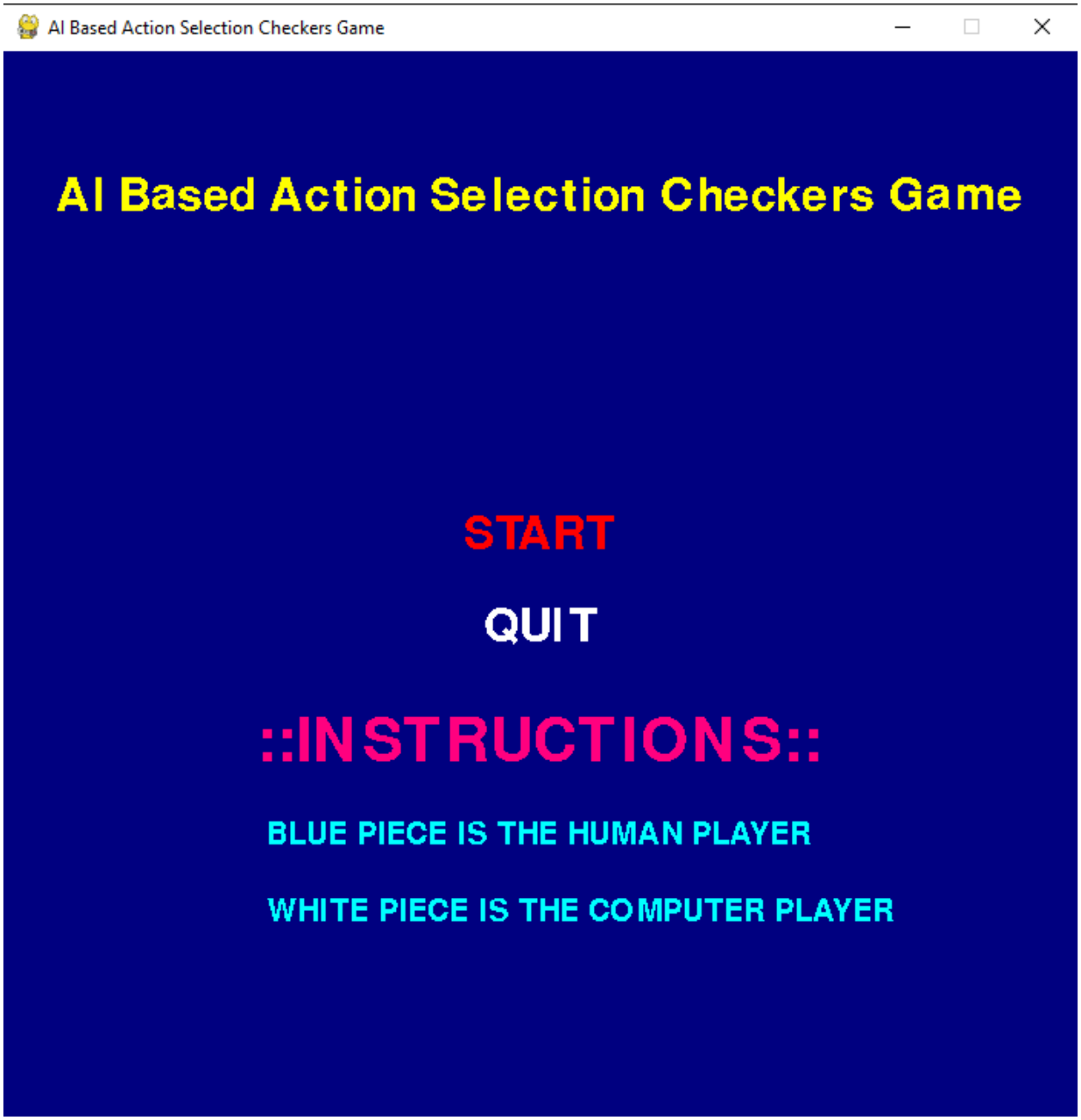


The human player plays the game first



Automatically the AI “WHITE” piece plays its turn

Date: 12-04-2021



The main menu screen is also finished.

When the user clicks on start the game starts and when the user selects Quit the game is closed automatically.

The instructions of the game are given below

FINAL CODE

Main.py

```
import pygame
from checkers.constants import WIDTH, HEIGHT, SQUARE_SIZE, RED, WHITE, BLUE
from checkers.board import Board
from checkers.game import Game
from minimax.algorithm import minimax

import pygame
from pygame.locals import *
import os

# Game Initialization
pygame.init()

# Center the Game Application
os.environ['SDL_VIDEO_CENTERED'] = '1'

# Game Resolution
screen_width=700
screen_height=700
screen=pygame.display.set_mode((screen_width, screen_height))

# Text Renderer
def text_format(message, textFont, textSize, textColor):
    newFont=pygame.font.Font(textFont, textSize)
    newText=newFont.render(message, 0, textColor)

    return newText


# Colors
white=(255, 255, 255)
black=(0, 0, 0)
gray=(50, 50, 50)
red=(255, 0, 0)
green=(85,107,47)
blue=(0,0,128)
yellow=(255, 255, 0)
pink =(255,0,127)
dblue= (0,255,255)

# Game Fonts
font = "freesansbold.ttf"
```

```
# Game Framerate
clock = pygame.time.Clock()
FPS=30

# Main Menu
def main_menu():

    menu=True
    selected="start"
    selected1="options"
    selected1="option"
    selected1="option1"

    while menu:
        for event in pygame.event.get():
            if event.type==pygame.QUIT:
                pygame.quit()
                quit()

            if event.type==pygame.KEYDOWN:

                if event.key==pygame.K_UP:
                    selected="start"

                elif event.key==pygame.K_DOWN:
                    selected="quit"

                elif event.key==pygame.K_LEFT:
                    selected1="options"
                elif event.key==pygame.K_LEFT:
                    selected1="option"
                elif event.key==pygame.K_LEFT:
                    selected1="option1"

                if event.key==pygame.K_RETURN:
                    if selected=="start":
                        main()
                    if selected=="quit":
                        pygame.quit()
                        quit()
                    if selected1=="options":
                        pass
                    if selected1=="option":
                        pass
                    if selected1=="option1":
                        pass
```

```
# Main Menu UI
screen.fill(blue)
title=text_format("AI Based Action Selection Checkers Game", font, 30, yellow)
if selected=="start":
   text_start=text_format("START", font, 30, red)
else:
   text_start = text_format("START", font, 30, white)

if selected=="quit":
   text_quit=text_format("QUIT", font, 30, red)
else:
   text_quit = text_format("QUIT", font, 30, white)

if selected1=="options":
   text_opt=text_format("::INSTRUCTIONS::", font, 40, pink)
else:
   text_opt = text_format("::INSTRUCTIONS::",font,40,pink)

if selected1=="option":
   text_opts=text_format("BLUE  IS THE HUMAN PLAYER", font, 20, dblue)
else:
   text_opts = text_format("BLUE PIECE IS THE HUMAN PLAYER",font,20,dblue)

if selected1=="option1":
   text_opt1=text_format("WHITE PIECE IS THE COMPUTER PLAYER", font, 20,dblue)
else:
   text_opt1 = text_format("WHITE IS THE COMPUTER PLAYER",font,20,dblue)

title_rect=title.get_rect()
start_rect=text_start.get_rect()
quit_rect=text_quit.get_rect()
opt_rect=text_opt.get_rect()
opts_rect=text_opts.get_rect()
opt1_rect=text_opts.get_rect()

# Main Menu Text
screen.blit(title, (screen_width/2 - (title_rect[2]/2), 80))
screen.blit(text_start, (screen_width/2 - (start_rect[2]/2), 300))
screen.blit(text_quit, (screen_width/2 - (quit_rect[2]/2), 360))
screen.blit(text_opt, (screen_width/2 - (opt_rect[2]/2), 430))
screen.blit(text_opts, (screen_width/2 - (opts_rect[2]/2), 500))
screen.blit(text_opt1, (screen_width/2 - (opt1_rect[2]/2), 550))
pygame.display.update()
clock.tick(FPS)
pygame.display.set_caption("AI Based Action Selection Checkers Game")
```

```
    #Initialize the Game



#creating a module around the checkrs game and then add an API on top of the game
# so that we can use it with an AI later on

#1 setup a pygame display where we will be drawing everything onto.
#2 setup a basic event loop: this will check if we press the mouse, if we press a certain key.
MAIN LOOP
#3 setup basic drawings, draw the board for the chessboard, draw the pieces

FPS = 60     #G=Frame per second

WIN = pygame.display.set_mode((WIDTH, HEIGHT))
#WIN: a constant value. WIDTH,HEIGHT: 2 Variables we will define(in constants.py)

#set a caption
pygame.display.set_caption('Checkers Game using Minimax Agent')



#this will take the position of our mouse, and based on the position of our mouse what row and
col we are in
def get_row_col_from_mouse(pos):
   x, y = pos   #this is gonna be a tuple that will have the x pos of our mouse and the y position of
our mouse
   #and based on the square size we can calcu very easily which row and col we are in
   row = y // SQUARE_SIZE
   col = x // SQUARE_SIZE
   return row, col


#SEE MOUSE BUTTON DOWN
#when we press the mouse down,
# we will get what row and col were in, we will select that piece and move that piece wherever
we want to move



#define a main funcion
```

```
def main():
    #create an event loop: this will run 'x' times per second which will check everything
    run = True

    #define a clock:define our game to run at a constant frame rate.
    #the clock will make sure that the main event loop wont run too fast or too slow.
    clock = pygame.time.Clock()

    #create a Baord obj
    #board = Board()

    #create a Game object
    game = Game(WIN)


    #for moving the pieces
    #piece = board.get_piece(0,1)
    #board.move_pieces(piece,4, 3)

    while run:
        clock.tick(FPS)

        #FOR AI MINIMAX
        if game.turn == WHITE:
            value, new_board = minimax(game.get_board(), 3, WHITE, game)
            game.ai_move(new_board)


        #for winner
        if game.winner() != None:
            print(game.winner())
            run = False

        #setup the basic event loop for pygame
        for event in pygame.event.get():
            if event.type == pygame.QUIT:    #this means we hit the red button on the top of the
screen
                run = False

            if event.type ==  pygame.MOUSEBUTTONDOWN:     #This means we pressed any
mouse on our mouse down
                pos = pygame.mouse.get_pos()
                row, col = get_row_col_from_mouse(pos)

                #call the select moethod from game.py
                #if game.turn == BLUE:
```

```
            game.select(row, col)

        game.update()

        #board.draw_squares(WIN)    #from board.py file, this function is called.
        #board.draw(WIN)            #for pieces, from board.py
        #pygame.display.update()

    pygame.quit()
main_menu()
main()
```

Constants.py

```
#Put all the constant values inside of this file

import pygame

#DEFINE THE HEIGHT AND WIDTH
WIDTH, HEIGHT = 700, 700   #800 pixels

#DEFINE ROWS AND COLS IN CHECKERS BOARD
ROWS, COLS = 8, 8

#HOW BIG IS ONE SQAURE OF THE CHECKER BOARD
SQUARE_SIZE = WIDTH//COLS

#DEFINE VARIABLE FOR COLOURS
#RGB COLOR
RED = (255, 0, 0)
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
BLUE = (0,255,255)
GREY = (128,128,128)
GREEN = (0, 153, 0)


#load the king image
#pygame.transform will resize the image
#44,25 is the resolution of the image, this reletively keeps the aspect ratio of the image
# and it makes it small enough to put it in the direct center so that we will put on or pieces
CROWN = pygame.transform.scale(pygame.image.load('assets/crown.png'), (44, 25))
```

Game.py

```
import pygame
from .constants import RED, WHITE, SQUARE_SIZE, GREY, BLUE, GREEN, CROWN

#when we make a new piece we need to pass what row its in, what column its in, and what color
```

it is

```
class Piece:

    PADDING = 15   #for radius FROM DRAW_PIECE()
    OUTLINE = 2   #for the outline  FROM DRAW_PIECE()



    def __init__(self, row, col, color):
        self.row = row
        self.col = col
        self.color = color
        self.king = False  #This will tell us, are we a King piece? that means we can jump
backwards.
        self.x = 0
        self.y = 0
        self.calc_position()

        #DOWN = POSITIVE
        #UP = NEGATIVE
        #if self.color == BLUE:        #Direction will be negative, we are going up(down pieces)
            #self.direction = -1    #what way are we going, positive or negative.
        #else:
            #self.direction = 1




    #This will calculate the x and y positions base on the row and column we are in.
    #We need to know what our x and y positions will be according to the square size.
    def calc_position(self):
        self.x = SQUARE_SIZE * self.col + SQUARE_SIZE //2
        #col=dealing with the x which is the horizontal axis, and
        # squaresize//2: we want the piece to be in the middle of the squarex and y pos
        self.y = SQUARE_SIZE * self.row + SQUARE_SIZE //2


    #This function will change the King variable
    def King_piece(self):
        self.king = True    #we will make this piece a king


    #we will draw, we will draw the actual piece itself
    def draw_piece(self, win):

        radius = SQUARE_SIZE // 2 - self.PADDING
```

```
        #draw an outline (LARGER CIRCLE)
        pygame.draw.circle(win, GREY, (self.x, self.y), radius + self.OUTLINE)

        #start by drawing a circle and outline so that we can see it  (SMALLER CIRCLE)
        pygame.draw.circle(win, self.color, (self.x, self.y), radius)
        #x, y pos=center of the circle
        #pick a radius: based on padding bw the edge of the square and the circle. define class vari a
the top.

        #for king piece
        if self.king:
            #blit():put some image on to the screen, or put some surface on the screen.pygame
method
            win.blit(CROWN, (self.x - CROWN.get_width()//2, self.y - CROWN.get_height()//2))
            #take the x and y radius and the crown image height and wdth so that it fits in the middle


    #for moving the pieces, defined even in board.py
    def move_pieces(self, row, col):
        #when we move a piece, then it will have a new row, the piece will be updated
        self.row = row
        self.col = col
        self.calc_position()   #tells us the x and y position of our piece should be, so we have to
recalculate that

    def __repr__(self):
        return str(self.color)
```

Piece.py

```
import pygame
from .constants import RED, WHITE, SQUARE_SIZE, GREY, BLUE, GREEN, CROWN

#when we make a new piece we need to pass what row its in, what column its in, and what color
it is
class Piece:

    PADDING = 15   #for radius FROM DRAW_PIECE()
    OUTLINE = 2   #for the outline  FROM DRAW_PIECE()



    def __init__(self, row, col, color):
        self.row = row
        self.col = col
        self.color = color
```

```
        self.king = False  #This will tell us, are we a King piece? that means we can jump
backwards.
        self.x = 0
        self.y = 0
        self.calc_position()

        #DOWN = POSITIVE
        #UP = NEGATIVE
        #if self.color == BLUE:        #Direction will be negative, we are going up(down pieces)
            #self.direction = -1    #what way are we going, positive or negative.
        #else:
            #self.direction = 1




    #This will calculate the x and y positions base on the row and column we are in.
    #We need to know what our x and y positions will be according to the square size.
    def calc_position(self):
        self.x = SQUARE_SIZE * self.col + SQUARE_SIZE //2
        #col=dealing with the x which is the horizontal axis, and
        # squaresize//2: we want the piece to be in the middle of the squarex and y pos
        self.y = SQUARE_SIZE * self.row + SQUARE_SIZE //2


    #This function will change the King variable
    def King_piece(self):
        self.king = True    #we will make this piece a king


    #we will draw, we will draw the actual piece itself
    def draw_piece(self, win):

        radius = SQUARE_SIZE // 2 - self.PADDING

        #draw an outline (LARGER CIRCLE)
        pygame.draw.circle(win, GREY, (self.x, self.y), radius + self.OUTLINE)

        #start by drawing a circle and outline so that we can see it  (SMALLER CIRCLE)
        pygame.draw.circle(win, self.color, (self.x, self.y), radius)
        #x, y pos=center of the circle
        #pick a radius: based on padding bw the edge of the square and the circle. define class vari a
the top.

        #for king piece
        if self.king:
```

```
        #blit():put some image on to the screen, or put some surface on the screen.pygame
method
        win.blit(CROWN, (self.x - CROWN.get_width()//2, self.y - CROWN.get_height()//2))
        #take the x and y radius and the crown image height and wdth so that it fits in the middle


    #for moving the pieces, defined even in board.py
    def move_pieces(self, row, col):
        #when we move a piece, then it will have a new row, the piece will be updated
        self.row = row
        self.col = col
        self.calc_position()   #tells us the x and y position of our piece should be, so we have to
recalculate that

    def __repr__(self):
        return str(self.color)
```

Board.py

```
#Put a class named "Board": This class will represent a checkers board.
#This class will handle all the different pieces moving, leading specific pieves, deleting specific
pieces, rawing itself on the screen.

import pygame
from .constants import BLACK, ROWS, RED, SQUARE_SIZE, COLS, WHITE, BLUE,
GREEN
from .piece import Piece


class Board:
    def __init__(self):

        #Attributes

        #1. Internal representation of the board
        self.board = []   #creating a 2D list
        #Whose turn is it
        #self.selected_piece = None  #have we selected a piece yet, or not
        self.red_left = self.white_left = 12 #keeps track of how many red, how many white pieces
we have.
        self.red_kings = self.white_kings = 0
        self.create_board()



    #A surface/window to draw the red and black cubes on in a checkerboard pattern
```

```
    def draw_squares(self, win):
        win.fill(BLACK)
        for row in range(ROWS):
            for col in range(row % 2, COLS, 2):
                pygame.draw.rect(win, RED,(row*SQUARE_SIZE, col*SQUARE_SIZE,
SQUARE_SIZE, SQUARE_SIZE
                )) #Drawing the red rectangle starting from the top left



    #for moving the pieces
    #sel, piece=which piece we want to move it to, row, col=on which row and col we want to
move it to
    def move_pieces(self, piece, row, col):
        #move the piece within the list, also chenge the piece itself
        self.board[piece.row][piece.col], self.board[row][col] = self.board[row][col],
self.board[piece.row][piece.col]
        #swapping is happening by reversing

        piece.move_pieces(row, col)

        #Checking: if the piece hits the last row or first col, then making that piece as a king.
        if row == ROWS -1 or row == 0:
            piece.King_piece()  #King_piece(): from piece file, this will make the piece a king

            if piece.color == WHITE:
                self.white_kings += 1

            else:
                self.red_kings += 1

    #giving the baord object a row and col, and it wll give a piece back
    def get_piece(self, row, col):
        return self.board[row][col]



    #PIECES, creating the actual internal representation of the board and we will add a bunch of
pieces to the list
    #A new class "piece.py"
    def create_board(self):
        for row in range(ROWS):
            self.board.append([])   #empty list: interior list for each row,
                                    #a list that represents what each row is going to have inside of it
```

```
        for col in range(COLS):

            #if the current column tha we are one, if its divisble by if that equals to row+1, so we
are on row 1
            if col % 2 == ((row + 1) % 2):   #part 2 3mins

                #draw when we are in a certain row.
                # if row < 3, 0 1 2 are the first 3 rows, we want to draw the white pieces in
                if row < 3:
                    self.board[row].append(Piece(row, col, WHITE))

                #if row>4, if irow is 5 6 7, then  we want to draw the red pieces
                elif row > 4:
                    self.board[row].append(Piece(row, col, BLUE))

                #when no piece, add a zero
                else:
                    # this will be a blank piece
                    self.board[row].append(0)
            #when we dont add a piece, add a zero
            else:
                self.board[row].append(0)

    #This will draw the board, this will draw all of the pieces and the squares.
    def draw(self, win):
        self.draw_squares(win)   #we will draw the sqaures on the window

        #Loop throuhout the pieces and draw those
        for row in range(ROWS):
            for col in range(COLS):

                #loop through the board
                piece = self.board[row][col]
                if piece != 0:     #if piece is 0, we will not draw anything
                    piece.draw_piece(win)    # draw the oiece on the window


    def remove(self, pieces):
        for piece in pieces:
            self.board[piece.row][piece.col] = 0
            if piece != 0:
                if piece.color == BLUE:
                    self.red_left -=1
                else:
                    self.white_left -= 1
```

```
    def winner(self):      #This will return the color if they won
        if self.red_left <= 0:
            return WHITE
        elif self.white_left <= 0:
            return BLUE

        return None

    def get_valid_moves(self, piece):
        moves = {}     #store the move as the key, so what place we could potentially move to as a
row

        #these are the diagonals (left or right)
        left = piece.col - 1   #we are moving left one
        right = piece.col + 1
        row = piece.row

        #checking whether we can move up or we can move down based on the color and based on
if the piece is a KING
        if piece.color == BLUE or piece.king:
            moves.update(self._traverse_left(row -1, max(row-3, -1), -1, piece.color, left))
            moves.update(self._traverse_right(row -1, max(row-3, -1), -1, piece.color, right))

        if piece.color == WHITE or piece.king:
            moves.update(self._traverse_left(row +1, min(row+3, ROWS), 1, piece.color, left))
            moves.update(self._traverse_right(row +1, min(row+3, ROWS), 1, piece.color, right))

        return moves

    #This function will look at the left diagonals for us.
    def _traverse_left(self, start, stop, step, color, left, skipped=[]):
        moves = {}
        last = []
        for r in range(start, stop, step):
            if left < 0:
                break   #when we are at the edge of the board on the left side then no further cols we
have

            current = self.board[r][left]
            if current == 0:
                if skipped and not last:
                    break
                elif skipped:
                    moves[(r, left)] = last + skipped

                else:
```

```
                moves[(r, left)] = last

            if last:
                if step == -1:
                    row = max(r-3, 0)
                else:
                    row = min(r+3, ROWS)

                moves.update(self._traverse_left(r+step, row, step, color, left-1, skipped=last))
                moves.update(self._traverse_right(r+step, row, step, color, left+1, skipped=last))
            break

        elif current.color == color:
            break
        else:
            last = [current]

        left -= 1

    return moves

  #WHY USE THESE PARAMETERS:
  #start, stop, step is going to be for the for loops we're going to put inside of here
  #step: very imp, becuase it tells us, we go up? or down? when im traversing thro the rows for the diagonals
  #skipped: this will tell us, have we skipped any pieces yet? if yes, we can only move to squares when we skip another piece

  #This method will move to the right
  #right: where are we starting in terms of col, when were traversing to the left
  def _traverse_right(self, start, stop, step, color, right, skipped=[]):
    moves = {}
    last = []
    for r in range(start, stop, step):
        if right >= COLS:
            break   #when we are at the edge of the board on the left side then no further cols we have

        current = self.board[r][right]
        if current == 0:
            if skipped and not last:
                break
            elif skipped:
                moves[(r, right)] = last + skipped

            else:
```

```
                moves[(r, right)] = last

            if last:
                if step == -1:
                    row = max(r-3, 0)
                else:
                    row = min(r+3, ROWS)

                moves.update(self._traverse_left(r+step, row, step, color, right-1, skipped=last))
                moves.update(self._traverse_right(r+step, row, step, color, right+1, skipped=last))
            break

        elif current.color == color:
            break
        else:
            last = [current]

        right += 1

    return moves

                                #FOR AI MINIMAX
    #The evaluate(): this method will tell us given the state of this board what is its score?
    def evaluate(self):   #This will take into account hwo many kings we have? and how many
pieces we have?
        return self.white_left - self.red_left + (self.white_kings * 0.5 - self.red_kings * 0.5)
        #Thsi will give us another value.

    #This method will return to us all the pieces of a certain color. so we can use the board
    def get_all_pieces(self, color):
        pieces = []
        for row in self.board:
            for piece in row:
                if piece != 0 and piece.color == color:
                    pieces.append(piece)
        return pieces
```

Algorithm.py

```
from copy import deepcopy
import pygame

BLUE = (0,255,255)
WHITE = (255, 255, 255)
RED= (255, 0, 0)
```

```
#position:- This stands for the current posisiton that we are in.
   #depth:- Tells us how how far or extendeting the tree.
     #This will be a recursive call. we will decrement this
   #max_player: A boolean value, which tells are we minimizing the value or min the value.
   #game:- The actual game object which we will get from main.py file and pass that to the alg

def minimax(position, depth, max_player, game):
   if depth == 0 or position.winner() != None:
      return position.evaluate(), position

   if max_player:
      maxEval = float('-inf')
      best_move = None
      for move in get_all_moves(position, WHITE, game):
         evaluation = minimax(move, depth-1, False, game)[0]
         maxEval = max(maxEval, evaluation)
         if maxEval == evaluation:
            best_move = move

      return maxEval, best_move

   else:
      minEval = float('inf')
      best_move = None
      for move in get_all_moves(position, BLUE, game):
         evaluation = minimax(move, depth-1, True, game)[0]
         minEval = min(minEval, evaluation)
         if minEval == evaluation:
            best_move = move

      return minEval, best_move


def simulate_move(piece, move, board, game, skip):
   board.move_pieces(piece, move[0], move[1])
   if skip:
      board.remove(skip)

   return board

def get_all_moves(board, color, game):
   moves = []

   for piece in board.get_all_pieces(color):
      valid_moves = board.get_valid_moves(piece)
```

```
        for move, skip in valid_moves.items():
            temp_board = deepcopy(board)
            temp_piece = temp_board.get_piece(piece.row, piece.col)
            new_board = simulate_move(temp_piece, move, temp_board, game, skip)
            moves.append(new_board)
    return moves
```