University of Asia Pacific

OTHELLO (REVERSI)

Repack by shoumik121

Submitted By

Shoumik Rouf
17101070(ID)
1st sem, 4th year
2020

Submitted To

Nadeem Ahmed
UAP
2020

Abstract:

Othello Master utilises the principles of game theory and implements them in an imperative programming language. Its main strength is its performance against other Othello-playing applications and it can be distinguished from typical such applications in the following aspects.

This report presents the implementation of othello or reversi game in py.game platform, hard coded with python only.

Assets are gathered from stock images, public to further use.

No audio was used.

Contents:

Introduction

The application will rely on principles of Game Theory, namely alpha-beta search and the minimax algorithm which will be explained later. Using these principles, the application is able to play the game at a very high level of competence, as well as provide a generic platform for games to be played on.

Game-playing programs have existed from the early years of computer science and have improved in their ability very rapidly. Game Theory was developed and established in the 1940's, making a contribution to the world of mathematics.

Some of the more prominent impacts that the topic has had is reflected in the field of economy where prediction of trends and human behaviour is fundamental. Programs that play well, on the other hand, served people's curiosity and set new challenges in the scene of world-class Chess.

Artificial Intelligence is now said to be tightly associated with such programs that can think. Computers, however, use brute-force to simulate human thinking, whereas real neural functions use associations, visual memory, etc.

It would be safe to say that computers have not yet reached this goal of being able to think. Brute-force may work for a game of Chess, but see Go for the very contrary.

Background Theory

The game Othello is considered to be a game that requires comprehensive experience to be mastered. It is also said that Othello *takes a minute to learn and a lifetime to master*. What makes it particularly interesting is the difficulty in telling which player is at a point of advantage until the late stages of the game. This is, in principle, where lack of skill and experience take their toll.

brief summary of the rules:

> At the beginning of the game, four stones are already placed at the centre of an 8x8 standard game board. Two stones of each one of the players are placed diagonally and, by convention, the player to make the first move is of white colour, whereas the other is of black colour. The stones are all two-sided and flipping them changes their colour to the opponent's colour. A legal move is such that it reverses one or more stones of the opponent's colour. To reverse a stone, a player places one of his/her stones in such a way so that it surrounds a sequence of one or more of the opponent's stones. Such a sequence of opponent's stones must be ending in a board slot that is occupied by the reversing player's stone. All straight lines are applicable in such a reversal: horizontal, vertical or diagonal. If no reversal is possible, the turn is passed to the opponent. The game is finished when neither player has any legal moves left. Usually, by this point the board is completely full. Whoever has the most stones placed on the board at that point wins the game.

Trees

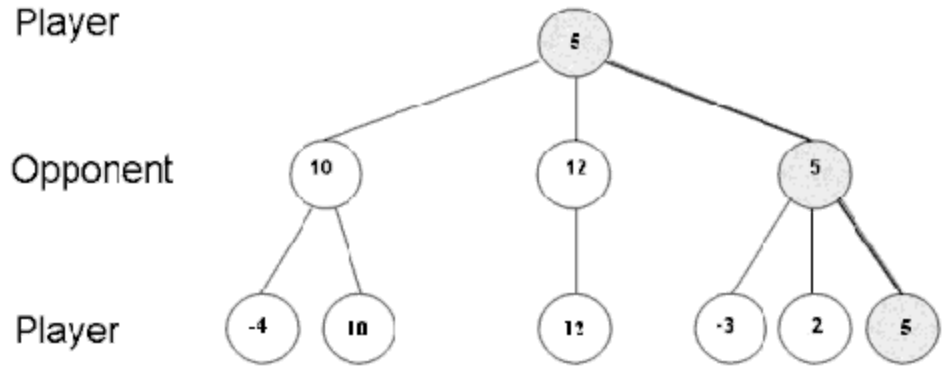
A game where decisions are made by a set of players can be described as a tree. At the start, there is one single state which is the opening state; no choice has yet been made by this point. That state corresponds to the initial board layout and in the case of Othello, the state is one where 4 stones are placed in the centre of the board. As the game progresses, a set of players can pull the state of the board in different directions, meaning that the state of the board at any point will depend on which *paths* in the tree have been picked up by the players in the game. In most board games, any state of the board is changed upon a placement or displacement of a game piece. A node in the tree, depicted below as a circle, presents one distinct board state that can be reached in one distinct way from the root. In other words, only one path can lead to a given node, hence forming a tree and not a graph. The nodes are the decision point in the game and combined nodes, connected by the relevant edges, form a path.

The minimax algorithm

The principles described here and onwards are strictly concerned with games where 2 players are involved and the information is complete. This means that the game position is always clearly known to both players. Also, due to the existence of only 2 players, from one player's point of view, any move made by his/her opponent has a direct influence on that one player.

In a game tree, we can incorporate the results obtained from the evaluation function. In other words, we can assign to each node of the tree the evaluation's returned value once it has been calculated. Since these game trees are usually enormous in size, the traversal to is limited to a certain depth. Trees generated can be depicted similarly to the one in Figure 3 below

Example:



Alpha-beta pruning

Alpha-beta pruning relies on the algorithm presented above. It is an extension that allows the process carried out by the minimax algorithms to be carried more wisely and explore more important paths in the game tree. It is based on the idea that branches in the game tree should no longer be explored if they offer a solution that is no better than what has already been found. The *pruning* of the tree means that we can leave out parts of the tree without needing to explore them. They simply do not offer any better choices than the ones already discovered. In order to allow for pruning to take place, the values of choices already explored need to be recorded as a range between two numbers. Alpha and beta were the original names for the variables holding these values and defining that range. The range indicates which values are still sought and it is modified as the tree is traversed. Making good traversal order allows for more pruning, but is a very difficult task. See the references for more details on the issues and implementation of alpha-beta pruning.

Evaluation Function, Strategies & Game Value

In order to look ahead in the game, one needs to choose an algorithm that will account for the usefulness of a move from the perspective of each player in the game. Another way of expressing this notion of usefulness is to talk about the *state evaluation* of a move. A simple example would be to describe a winning move as one that carries a high value.

A strategy defines a choice of a path in the game tree. It can be seen as the tendency adopted by one player, choosing to stand when having a sum of 20 in Blackjack. The focus of game-playing programs development is the discovery of good strategies.

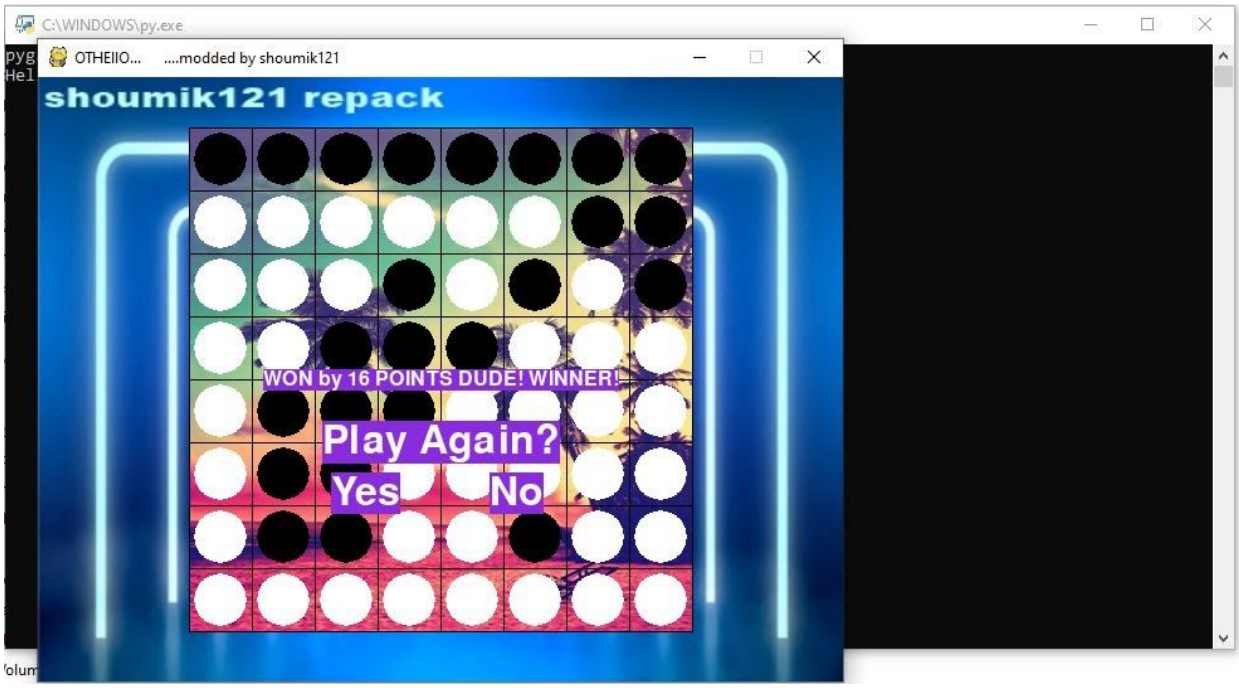
The game value indicates what the pay-off for each player is when the game is orchestrated by optimal strategies. It can indicate if one player can ensure at most a win, a draw or a loss. Within the population of large games, we usually do not know which of these categories a game falls into because the game tree is extraordinarily wide. Othello is one of these games.

Design

Applying open libraries,
allowing for opening libraries to be stored and retrieved from very quickly. It is an ad-hoc facility that comprises the functions which are invoked by the program only. Small redundancy only exists where the program's extensions may require it. This is the only unit that includes segments of code.
Use of stock images for backgrounds and basic colours.

Background of the board:

Over all:



Implementation

Coding of the required application was in the imperative language python. Object-oriented languages were not the most convenient choice due to the nature of the given py.game libraries. The Makefile was constructed by Mr. Toby Howard and was made available to any departmental client of the py.game libraries. The destination platform was windows. Attempted porting to Windows was obstructed due to the absence of the corresponding pygame libraries in the department.

Some specific aspects of the implementation are worth mentioning more than others. While some of the functionality coded may be either trivial or too general to be argued about, there are certain points that not only are less trivial, but also provide a good overview on the operation of this particular game-playing program.

It would therefore be wiser choice to focus on the operations and procedures associated with the game engine. Also, some of the related key issues that need to be addressed in order to support this engine will be covered. This section illustrates some of the implementation issues of the supporting components and the subsequent section explains some of the playing-engine components and properties.

Inevitably, summarising the full structure and game features of the package is beyond the scope of these reports. In particular, some of the low and intermediate level implementation issues will be left out. Instead, assorted issues will be specified and analysed. A full picture can only be obtained by reading through the enclosed source code.

Source Code

```
import random, sys, pygame, time, copy
from pygame.locals import *

FPS = 10
WINDOWWIDTH = 640
WINDOWHEIGHT = 480
SPACESIZE = 50
BOARDWIDTH = 8
BOARDHEIGHT = 8

XMARGIN = int((WINDOWWIDTH - (BOARDWIDTH * SPACESIZE)) / 2)

YMARGIN = int((WINDOWHEIGHT - (BOARDHEIGHT * SPACESIZE)) / 2)

WHITE_TILE = 'WHITE_TILE'
BLACK_TILE = 'BLACK_TILE'
EMPTY_SPACE = 'EMPTY_SPACE'
HINT_TILE = 'HINT_TILE'
ANIMATIONSPEED = 25




#               R    G    B
WHITE      = (255, 255, 255)
BLACK      = (  0,   0,   0)
GREEN      = (  0, 155,   0)
BRIGHTBLUE = (  0,  50, 255)
BROWN      = (174,  94,   0)
RED        = (255,   0,   0)
fuchsia    = (255,   0, 255)
blueviolet = (138,  43, 226)

TEXTBGCOLOR1 = blueviolet
TEXTBGCOLOR2 = RED
GRIDLINECOLOR = BLACK
TEXTCOLOR = WHITE
HINTCOLOR = fuchsia


def main():
   global MAINCLOCK, DISPLAYSURF, FONT, BIGFONT, BGIMAGE

   pygame.init()
   MAINCLOCK = pygame.time.Clock()
   DISPLAYSURF = pygame.display.set_mode((WINDOWWIDTH, WINDOWHEIGHT))
   pygame.display.set_caption('OTHEllO...      ....modded by shoumik121')
   FONT = pygame.font.Font('freesansbold.ttf', 16)
```

```
    BIGFONT = pygame.font.Font('freesansbold.ttf', 32)


    boardImage = pygame.image.load('flippyboard.png')

    boardImage = pygame.transform.smoothscale(boardImage, (BOARDWIDTH *
SPACESIZE, BOARDHEIGHT * SPACESIZE))
    boardImageRect = boardImage.get_rect()
    boardImageRect.topleft = (XMARGIN, YMARGIN)
    BGIMAGE = pygame.image.load('flippybackground.png')
    # smoothscale() background image to fit the entire window:
    BGIMAGE = pygame.transform.smoothscale(BGIMAGE, (WINDOWWIDTH,
WINDOWHEIGHT))
    BGIMAGE.blit(boardImage, boardImageRect)


    while True:
        if runGame() == False:
            break


def runGame():


    # Reset the board and game.
    mainBoard = getNewBoard()
    resetBoard(mainBoard)
    showHints = False
    turn = random.choice(['AI', 'Human'])


    drawBoard(mainBoard)
    playerTile, computerTile = enterPlayerTile()

    # Make the Surface and Rect objects for the "New Game" and "Hints"
buttons
    newGameSurf = FONT.render('New Game', True, TEXTCOLOR, TEXTBGCOLOR2)
    newGameRect = newGameSurf.get_rect()
    newGameRect.topright = (WINDOWWIDTH - 8, 10)
    hintsSurf = FONT.render('Hints', True, TEXTCOLOR, TEXTBGCOLOR2)
    hintsRect = hintsSurf.get_rect()
    hintsRect.topright = (WINDOWWIDTH - 8, 40)

    while True: #GL
        # L'n till Ai's turn
        if turn == 'HUMAN':
            # Human
            if getValidMoves(mainBoard, playerTile) == []:

                # can't move by human then end the game.
                break
            movexy = None
            while movexy == None:
                # Keep l'n until the player clicks on a valid space.
```

```
                if showHints:
                    boardToDraw = getBoardWithValidMoves(mainBoard, playerTile)
                else:
                    boardToDraw = mainBoard

                checkForQuit()
                for event in pygame.event.get(): # event handling l
                    if event.type == MOUSEBUTTONUP:
                        # Handle mouse click events
                        mousex, mousey = event.pos
                        if newGameRect.collidepoint( (mousex, mousey) ):
                            # Start a new game
                            return True
                        elif hintsRect.collidepoint( (mousex, mousey) ):
                            # Toggle hints mode
                            showHints = not showHints

                        movexy = getSpaceClicked(mousex, mousey)
                        if movexy != None and not isValidMove(mainBoard,
playerTile, movexy[0], movexy[1]):
                            movexy = None


                drawBoard(boardToDraw)
                drawInfo(boardToDraw, playerTile, computerTile, turn)

                # Draw the "New Game" and "Hints" buttons.
                DISPLAYSURF.blit(newGameSurf, newGameRect)
                DISPLAYSURF.blit(hintsSurf, hintsRect)

                MAINCLOCK.tick(FPS)
                pygame.display.update()

            # Make the move and end the turn.
            makeMove(mainBoard, playerTile, movexy[0], movexy[1], True)
            if getValidMoves(mainBoard, computerTile) != []:
                # Only set for the Ai's turn if it can make a move.
                turn = 'AI'

        else:
            # Ai's turn:
            if getValidMoves(mainBoard, computerTile) == []:

                # if AI can't move, then end the game.
                break


            drawBoard(mainBoard)
            drawInfo(mainBoard, playerTile, computerTile, turn)

            # Draw the "New Game" and "Hints" buttons.
            DISPLAYSURF.blit(newGameSurf, newGameRect)
            DISPLAYSURF.blit(hintsSurf, hintsRect)

            # Pause Illusion
            pauseUntil = time.time() + random.randint(5, 15) * 0.1
```

```
            while time.time() < pauseUntil:
                pygame.display.update()

            # Make the move and end the turn.
            x, y = getComputerMove(mainBoard, computerTile)
            makeMove(mainBoard, computerTile, x, y, True)
            if getValidMoves(mainBoard, playerTile) != []:
                # Only set for the player's turn if they can make a move.
                turn = 'HUMAN'

    # Display the final score.
    drawBoard(mainBoard)
    scores = getScoreOfBoard(mainBoard)


    if scores[playerTile] > scores[computerTile]:
        text = 'WON by %s POINTS DUDE! WINNER!' % \
               (scores[playerTile] - scores[computerTile])
    elif scores[playerTile] < scores[computerTile]:
        text = 'LOOSER. LOST BY %s POINTS.' % \
               (scores[computerTile] - scores[playerTile])
    else:
        text = 'The GAMER waz a TIE!'

    textSurf = FONT.render(text, True, TEXTCOLOR, TEXTBGCOLOR1)
    textRect = textSurf.get_rect()
    textRect.center = (int(WINDOWWIDTH / 2), int(WINDOWHEIGHT / 2))
    DISPLAYSURF.blit(textSurf, textRect)

    # Display Play again?
    text2Surf = BIGFONT.render('Play Again?', True, TEXTCOLOR, TEXTBGCOLOR1)
    text2Rect = text2Surf.get_rect()
    text2Rect.center = (int(WINDOWWIDTH / 2), int(WINDOWHEIGHT / 2) + 50)

    #"Yes" button.
    yesSurf = BIGFONT.render('Yes', True, TEXTCOLOR, TEXTBGCOLOR1)
    yesRect = yesSurf.get_rect()
    yesRect.center = (int(WINDOWWIDTH / 2) - 60, int(WINDOWHEIGHT / 2) + 90)

    #"No" button.
    noSurf = BIGFONT.render('No', True, TEXTCOLOR, TEXTBGCOLOR1)
    noRect = noSurf.get_rect()
    noRect.center = (int(WINDOWWIDTH / 2) + 60, int(WINDOWHEIGHT / 2) + 90)

    while True:

        checkForQuit()
        for event in pygame.event.get():
            if event.type == MOUSEBUTTONUP:
                mousex, mousey = event.pos
                if yesRect.collidepoint( (mousex, mousey) ):
                    return True
                elif noRect.collidepoint( (mousex, mousey) ):
                    return False
        DISPLAYSURF.blit(textSurf, textRect)
        DISPLAYSURF.blit(text2Surf, text2Rect)
```

```
        DISPLAYSURF.blit(yesSurf, yesRect)
        DISPLAYSURF.blit(noSurf, noRect)
        pygame.display.update()
        MAINCLOCK.tick(FPS)


def translateBoardToPixelCoord(x, y):
   return XMARGIN + x * SPACESIZE + int(SPACESIZE / 2), YMARGIN + y *
SPACESIZE + int(SPACESIZE / 2)


def animateTileChange(tilesToFlip, tileColor, additionalTile):


   if tileColor == WHITE_TILE:
       additionalTileColor = WHITE
   else:
       additionalTileColor = BLACK
   additionalTileX, additionalTileY =
translateBoardToPixelCoord(additionalTile[0], additionalTile[1])
   pygame.draw.circle(DISPLAYSURF, additionalTileColor, (additionalTileX,
additionalTileY), int(SPACESIZE / 2) - 4)
   pygame.display.update()

   for rgbValues in range(0, 255, int(ANIMATIONSPEED * 2.55)):
       if rgbValues > 255:
           rgbValues = 255
       elif rgbValues < 0:
           rgbValues = 0

       if tileColor == WHITE_TILE:
           color = tuple([rgbValues] * 3)
       elif tileColor == BLACK_TILE:
           color = tuple([255 - rgbValues] * 3)

       for x, y in tilesToFlip:
           centerx, centery = translateBoardToPixelCoord(x, y)
           pygame.draw.circle(DISPLAYSURF, color, (centerx, centery),
int(SPACESIZE / 2) - 4)
       pygame.display.update()
       MAINCLOCK.tick(FPS)
       checkForQuit()


def drawBoard(board):

   DISPLAYSURF.blit(BGIMAGE, BGIMAGE.get_rect())


   for x in range(BOARDWIDTH + 1):

       startx = (x * SPACESIZE) + XMARGIN
       starty = YMARGIN
       endx = (x * SPACESIZE) + XMARGIN
       endy = YMARGIN + (BOARDHEIGHT * SPACESIZE)
       pygame.draw.line(DISPLAYSURF, GRIDLINECOLOR, (startx, starty), (endx,
```

```
endy))
   for y in range(BOARDHEIGHT + 1):

       startx = XMARGIN
       starty = (y * SPACESIZE) + YMARGIN
       endx = XMARGIN + (BOARDWIDTH * SPACESIZE)
       endy = (y * SPACESIZE) + YMARGIN
       pygame.draw.line(DISPLAYSURF, GRIDLINECOLOR, (startx, starty), (endx,
endy))


   for x in range(BOARDWIDTH):
       for y in range(BOARDHEIGHT):
           centerx, centery = translateBoardToPixelCoord(x, y)
           if board[x][y] == WHITE_TILE or board[x][y] == BLACK_TILE:
               if board[x][y] == WHITE_TILE:
                   tileColor = WHITE
               else:
                   tileColor = BLACK
               pygame.draw.circle(DISPLAYSURF, tileColor, (centerx, centery),
int(SPACESIZE / 2) - 4)
           if board[x][y] == HINT_TILE:
               pygame.draw.rect(DISPLAYSURF, HINTCOLOR, (centerx - 4, centery
- 4, 8, 8))


def getSpaceClicked(mousex, mousey):


   for x in range(BOARDWIDTH):
       for y in range(BOARDHEIGHT):
           if mousex > x * SPACESIZE + XMARGIN and \
              mousex < (x + 1) * SPACESIZE + XMARGIN and \
              mousey > y * SPACESIZE + YMARGIN and \
              mousey < (y + 1) * SPACESIZE + YMARGIN:
               return (x, y)
   return None


def drawInfo(board, playerTile, computerTile, turn):

   scores = getScoreOfBoard(board)
   scoreSurf = FONT.render("HUMANS(score): %s
AIZ(score): %s                          %s It's Your Turn" %
(str(scores[playerTile]), str(scores[computerTile]), turn.title()), True,
TEXTCOLOR)
   scoreRect = scoreSurf.get_rect()
   scoreRect.bottomleft = (10, WINDOWHEIGHT - 5)
   DISPLAYSURF.blit(scoreSurf, scoreRect)


def resetBoard(board):

   for x in range(BOARDWIDTH):
       for y in range(BOARDHEIGHT):
           board[x][y] = EMPTY_SPACE
```

```
    # Add starting pieces to the center
    board[3][3] = WHITE_TILE
    board[3][4] = BLACK_TILE
    board[4][3] = BLACK_TILE
    board[4][4] = WHITE_TILE


def getNewBoard():
    # new, empty board data structure.
    board = []
    for i in range(BOARDWIDTH):
        board.append([EMPTY_SPACE] * BOARDHEIGHT)

    return board


def isValidMove(board, tile, xstart, ystart):
    # Returns False if the player's move is invalid. If it is a valid
    # move, returns a list of spaces of the captured pieces.
    if board[xstart][ystart] != EMPTY_SPACE or not isOnBoard(xstart, ystart):
        return False

    board[xstart][ystart] = tile

    if tile == WHITE_TILE:
        otherTile = BLACK_TILE
    else:
        otherTile = WHITE_TILE

    tilesToFlip = []
    # check each of the eight directions:
    for xdirection, ydirection in [[0, 1], [1, 1], [1, 0], [1, -1], [0, -1],
[-1, -1], [-1, 0], [-1, 1]]:
        x, y = xstart, ystart
        x += xdirection
        y += ydirection
        if isOnBoard(x, y) and board[x][y] == otherTile:
            # The piece belongs to the other player next to our piece.
            x += xdirection
            y += ydirection
            if not isOnBoard(x, y):
                continue
            while board[x][y] == otherTile:
                x += xdirection
                y += ydirection
                if not isOnBoard(x, y):
                    break # break out of while loop, continue in for loop
            if not isOnBoard(x, y):
                continue
            if board[x][y] == tile:
                # There are pieces to flip over. Go in the reverse
                # direction until we reach the original space, noting all
                # the tiles along the way.
                while True:
                    x -= xdirection
```

```
                    y -= ydirection
                    if x == xstart and y == ystart:
                        break
                    tilesToFlip.append([x, y])

    board[xstart][ystart] = EMPTY_SPACE # make space empty
    if len(tilesToFlip) == 0: # If no tiles flipped, this move is invalid
        return False
    return tilesToFlip


def isOnBoard(x, y):
    # Returns True if the coordinates are located on the board.
    return x >= 0 and x < BOARDWIDTH and y >= 0 and y < BOARDHEIGHT


def getBoardWithValidMoves(board, tile):
    # Returns a new board with hint markings.
    dupeBoard = copy.deepcopy(board)

    for x, y in getValidMoves(dupeBoard, tile):
        dupeBoard[x][y] = HINT_TILE
    return dupeBoard


def getValidMoves(board, tile):
    # Returns a list of (x,y) tuples of all valid moves.
    validMoves = []

    for x in range(BOARDWIDTH):
        for y in range(BOARDHEIGHT):
            if isValidMove(board, tile, x, y) != False:
                validMoves.append((x, y))
    return validMoves


def getScoreOfBoard(board):

    xscore = 0
    oscore = 0
    for x in range(BOARDWIDTH):
        for y in range(BOARDHEIGHT):
            if board[x][y] == WHITE_TILE:
                xscore += 1
            if board[x][y] == BLACK_TILE:
                oscore += 1
    return {WHITE_TILE:xscore, BLACK_TILE:oscore}


def enterPlayerTile():

    textSurf = FONT.render('Which One Do You Prefer White or Black?', True,
TEXTCOLOR, TEXTBGCOLOR1)
    textRect = textSurf.get_rect()
    textRect.center = (int(WINDOWWIDTH / 2), int(WINDOWHEIGHT / 2))
```

```
    xSurf = BIGFONT.render('WHITE', True, TEXTCOLOR, TEXTBGCOLOR1)
    xRect = xSurf.get_rect()
    xRect.center = (int(WINDOWWIDTH / 2) - 60, int(WINDOWHEIGHT / 2) + 40)

    oSurf = BIGFONT.render('BLACK', True, TEXTCOLOR, TEXTBGCOLOR1)
    oRect = oSurf.get_rect()
    oRect.center = (int(WINDOWWIDTH / 2) + 60, int(WINDOWHEIGHT / 2) + 40)

    while True:

        checkForQuit()
        for event in pygame.event.get():
            if event.type == MOUSEBUTTONUP:
                mousex, mousey = event.pos
                if xRect.collidepoint( (mousex, mousey) ):
                    return [WHITE_TILE, BLACK_TILE]
                elif oRect.collidepoint( (mousex, mousey) ):
                    return [BLACK_TILE, WHITE_TILE]


        DISPLAYSURF.blit(textSurf, textRect)
        DISPLAYSURF.blit(xSurf, xRect)
        DISPLAYSURF.blit(oSurf, oRect)
        pygame.display.update()
        MAINCLOCK.tick(FPS)


def makeMove(board, tile, xstart, ystart, realMove=False):

    tilesToFlip = isValidMove(board, tile, xstart, ystart)

    if tilesToFlip == False:
        return False

    board[xstart][ystart] = tile

    if realMove:
        animateTileChange(tilesToFlip, tile, (xstart, ystart))

    for x, y in tilesToFlip:
        board[x][y] = tile
    return True


def isOnCorner(x, y):

    return (x == 0 and y == 0) or \
           (x == BOARDWIDTH and y == 0) or \
           (x == 0 and y == BOARDHEIGHT) or \
           (x == BOARDWIDTH and y == BOARDHEIGHT)


def getComputerMove(board, computerTile):


    possibleMoves = getValidMoves(board, computerTile)
```

```
    random.shuffle(possibleMoves)


    for x, y in possibleMoves:
        if isOnCorner(x, y):
            return [x, y]


    bestScore = -1
    for x, y in possibleMoves:
        dupeBoard = copy.deepcopy(board)
        makeMove(dupeBoard, computerTile, x, y)
        score = getScoreOfBoard(dupeBoard)[computerTile]
        if score > bestScore:
            bestMove = [x, y]
            bestScore = score
    return bestMove


def checkForQuit():
    for event in pygame.event.get((QUIT, KEYUP)):
        if event.type == QUIT or (event.type == KEYUP and event.key ==
K_ESCAPE):
            pygame.quit()
            sys.exit()


if __name__ == '__main__':
    main()
```

Performance analysis

The following figure presents the values that were made permanent in the header file once the fine-tuning process had been completed. The value in each of the board slots indicates the significance of holding that position.

10k	-3k	1k	800	800	1k	-3k	10k
-3k	-5k	-450	-500	-500	-450	-5k	-3k
1k	-450	30	10	10	30	-450	1k
800	-500	10	50	50	10	-500	800
800	-500	10	50	50	10	-500	800
1k	-450	30	10	10	30	-450	1k
-3k	-5k	-450	-500	-500	-450	-5k	-3k
10k	-3k	1k	800	300	1k	-3k	10k

Conclusion

The project implemented and employed algorithms sufficiently strong to perform convincingly well against other available applications and has managed to analyse its different approaches to conclude which ones surpass all others and why. The project was constructed in accordance with the software engineering life-cycle and possible extensions have been proposed.

The project may contribute and assist others as it lays out successful approaches for artificially playing this game as well as bad approaches.