REVERSI GAME USING MINIMAX ALGORITHM

A PROJECT REPORT

Submitted by

MADHULIKA G (2116210701139)

MAHALAKSHMI K (2116210701143)

MERCY N (2116210701157)

in partial fulfillment for the award of the degree

of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING

RAJALAKSHMI ENGINEERING COLLEGE

ANNA UNIVERSITY, CHENNAI

NOVEMBER 2023

RAJALAKSHMI ENGINEERING COLLEGE, CHENNAI

BONAFIDE CERTIFICATE

Certified that this Thesis titled **“REVERSI GAME USING MINIMAX ALGORITHM**” is the bonafide work of “**MADHULIKA G (2116210701139),MAHALAKSHMI K (2116210701143), MERCY N (2116210701157)”** who carried out the work under my supervision. Certified further that to the best of my knowledge the work reported herein does not form part of any other thesis or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

SIGNATURE

Mrs. Susmita Mishra

SUPERVISOR

Assistant Professor (SG)

Department of Computer Science

and Engineering

Rajalakshmi Engineering College

Chennai - 602 105

Submitted to Project Viva-Voce Examination held on ___________________

Internal Examiner **External Examiner**

ABSTRACT

Reversi is a strategy board game for two players, played on an 8×8 board. Two players compete, using 64 identical game pieces ("disks") that are light on one side and dark on the other. Each player chooses one color to use throughout the game. Players take turns placing one disk on an empty square, with their assigned color facing up. After a play is made, any disks of the opponent's color that lie in a straight line bounded by the one just played and another one in the current player's color are turned over. When all playable empty squares are filled, the player with more disks showing in their own color wins the game. Algorithms for AI solutions come in many different forms and sizes. The Minimax algorithm is the hardest to defeat because it is a recursive or backtracking algorithm which is used in decision-making and game theory. This algorithm provides an optimal move for the player assuming that the opponent is also playing optimally. This algorithm includes two players in the game, one is MAX and the other is MIN, where they will fight for their maximum benefit while the opponent is getting the minimum benefit. The MAX will select the maximized value whereas the MIN will select the minimized value. This algorithm uses Depth-First Search algorithm for the exploration of the complete game tree. This Minimax algorithm is complete, optimal, and has the time and space complexities as O(b^m) and O(bm) respectively. The AI solution has different degrees; they are random, defensive and aggressive. Random, as the name implies, randomly picks a play and is the easiest to beat. Defensive AI makes blocking a win a priority, while aggressive AI makes winning a priority. Both are harder to beat than the random AI. The AI should evaluate the current state of the board to determine the best move, recursively search through possible moves and their outcomes, and assign a score to each possible move and choose the one with the highest score. When the game ends, display the result as win, lose or draw.

ACKNOWLEDGEMENT

First, we thank the almighty god for the successful completion of the project. Our sincere thanks to our chairman **Mr. S. Meganathan B.E., F.I.E.,** for his sincere endeavor in educating us in his premier institution. We would like to express our deep gratitude to our beloved Chairperson **Dr. Thangam Meganathan Ph.D.,** for her enthusiastic motivation which inspired us a lot in completing this project and Vice Chairman **Mr. Abhay Shankar Meganathan B.E., M.S.,** for providing us with the requisite infrastructure.

We also express our sincere gratitude to our college Principal, **Dr. S. N. Murugesan M.E., PhD.,** and **Dr. P. KUMAR M.E., PhD, Director computing and information science , and Head Of Department of Computer Science and Engineering** and our project guide **Mrs. Susmita Mishra MTech.,** for her encouragement and guiding us throughout the project towards successful completion of this project and to our parents, friends, all faculty members and supporting staffs for their direct and indirect involvement in successful completion of the project for their encouragement and support.

MADHULIKA G

MAHALAKSHMI K

MERCY N

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Artificial Intelligence is also being used in game design to create more dynamic and interesting levels and content. This can help developers create more diverse and engaging games with less effort. For example, AI might be used to design game levels that are procedurally generated, meaning that they are created on the fly as the player progresses through the game. This can help keep the game fresh and interesting for players, as they are not simply playing through the same levels over and over again.

Reversi is a two player game in which opponents takes turns placing coins of their color (white/black) onto the board, traditionally an 8 × 8 square grid. A legal move consists of a coin placement to an empty square which traps a continuous sequence (either horizontally, vertically, or diagonally) of your opponent's pieces between two coins of your color. All pieces which have become trapped in this manner are then flipped to the opposing color. If a player has no legal moves, they must pass, allowing their opponent to play again. The game begins with a board configuration consisting of 4 coins, 2 of each color placed diagonally, forming a square at the center of the board. Players then alternate turns until the board is full or neither player has a legal move. At this point, the player with the most coins of their color on the board is the winner.

The Minimax algorithm has become a standard feature of computer game players. Minimax is used to identify the best moves in a game tree generated by each player's legal actions. Terminal nodes represent finished games; these are scored according to the game rules. Scores can then be propagated backwards through the game tree by assuming that each player plays perfectly, always choosing the action which maximizes their eventual endgame score. For short games this algorithm can be used exactly, essentially solving the game. For longer games however, the full search is infeasible, and the tree search is usually halted at some fixed depth. Because these no longer correspond to finished games, a method for scoring intermediate states is required. For board games like checkers, chess, and Reversi, the number of intermediate states is immense, one for each unique board state, and a model for evaluating board positions is required. Much of the work in game AI centers on creating a good position evaluator - finding important features and designing or training a model to predict the quality of intermediate board positions.

Pygame is a cross-platform set of Python modules designed for writing video games. It is utilized for creating a visually appealing and user-friendly interface. Pygame is used for handling graphics, drawing the game grid, tokens, and transition animations. The game includes start and end screens with options for starting a new game after it's over.

1.1 PROBLEM STATEMENT

AI approached the problem of board evaluation primarily as a modeling challenge - identifying the useful features and selecting the correct model.

Minimax method is its independence from expert knowledge. All board evaluations are drawn (indirectly) from endgame scoring, rather than relying on potentially imperfect information. Developing a computer Reversi player requires a method for identifying the best move from any board position. Given unlimited computational resources, the ideal approach would be to exhaustively search the entire game tree, assuming both players play perfectly. The leaf nodes of the game tree represent endgame boards and can be scored for the white player as white coins − black coins. Non-terminal nodes represent boards where the active player has one or more legal moves. The assumption of perfect play implies that these nodes can be scored as the maximum (minimum) score of their child nodes for white (black) active player. By propagating the scores back to the starting position, the computer player can then choose the move which gives the best score. This is known as the Minimax algorithm and it has become a standard approach to game AI.

1.2 SCOPE OF THE WORK

The scope of the system is using Minimax algorithm to find the best available moves. Given unlimited computational resources, the ideal approach would be to exhaustively search the entire game tree, assuming both players play perfectly. The leaf nodes of the game tree represent endgame boards. Non-terminal nodes represent boards where the active player has one or more legal moves.

1.3 AIM AND OBJECTIVES OF THE PROJECT

The main aim of the Reversi game is to find the best available move by using the minimax algorithm. After each move the score of each player will be updated. Once the board is filled the player who has the maximum score will be the winner.

Alpha beta pruning optimize the Minimax algorithm by avoiding unnecessary exploration of certain branches in the game tree. This helps improve the efficiency of the AI's decision-making process.

Appealing user interface is required to improve the user experience. Using pygame the user interface for the board game was created. The coins are also designed using it.

1.4 RESOURCES

This project has been developed through widespread secondary research of accredited manuscripts, standard papers, white papers, analysts' information, and conference reviews. Significant resources are required to achieve an efficacious completion of this project.

The following prospectus details a list of resources that will play a primary role in the successful execution of our project:

>.A properly functioning workstation (PC, laptop, net-books etc.) to carry out desired research and collect relevant content.

>. Unlimited internet access.

>.The official Pygame documentation for detailed information on Pygame's modules and functions.

1.5 MOTIVATION

The motivation lies in the challenge of crafting a digital representation of the classic Othello board game, complete with an intelligent AI opponent employing the sophisticated Minimax algorithm. This project offers an opportunity to apply the knowledge of Minimax algorithm and alpha beta pruning. The user interface of the system was created using pygame which increases the user experience.

CHAPTER 2

LITERATURE SURVEY

2.1 SURVEY

Explore the use of AI in game development, specifically focusing on board games like Reversi. Discuss how AI algorithms enhance gameplay, including Minimax, Alpha-Beta Pruning, and heuristic evaluation functions. Survey existing research papers, articles, and academic works discussing the implementation of the Minimax algorithm in various board games, including Reversi. Understand the variations, optimizations, and challenges in applying Minimax. Review research on Alpha-Beta Pruning, a technique to enhance Minimax by reducing unnecessary exploration of the game tree. Understand its implementation and impact on improving the efficiency of decision-making in game AI. Explore resources and documentation on Pygame, focusing on its capabilities for creating user-friendly interfaces, handling graphics, animations, and user interaction. Understand how it can be utilized for developing the Reversi game.

2.2 PROPOSED SYSTEM

Minimax Algorithm Implementation: Discuss various approaches to implement the Minimax algorithm in Reversi, considering the evaluation function, tree depth, and computational resources. Highlight challenges and trade-offs in implementing the algorithm.

Alpha-Beta Pruning Integration: Explore ways to integrate Alpha-Beta Pruning into the Minimax algorithm for Reversi. Discuss how it optimizes the decision-making process and improves computational efficiency.

User Interface Development with Pygame: Detail the process of using Pygame for creating the Reversi game's user interface. Discuss strategies for displaying the game board, tokens, transitions, start/end screens, and user interaction.

2.3 INFERENCE MECHANISM

Evaluation Function Design: Discuss approaches to designing an effective evaluation function for Reversi. Explore heuristic strategies, positional evaluation, mobility, and other factors influencing board state assessment.

Handling Large Game Trees: Address challenges in handling large game trees in Reversi due to its complexity. Discuss methods to optimize the algorithm for better performance without compromising accuracy.

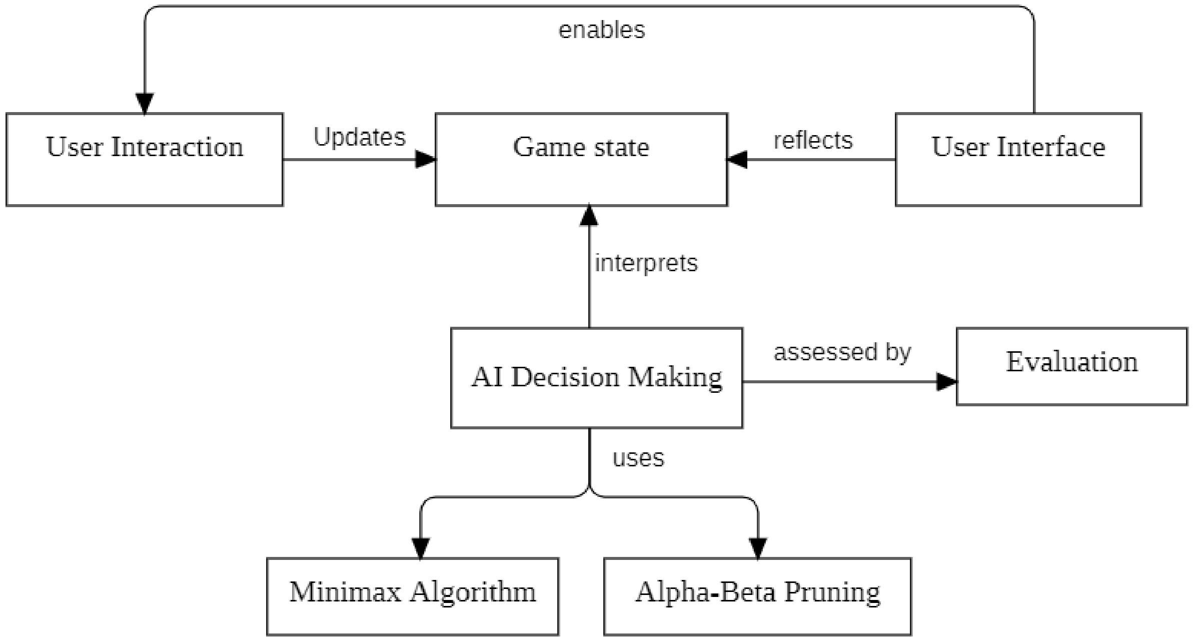

Fig 2.3 Inference diagram

2.4 MACHINE LEARNING APPROACHES IN BOARD GAME AI:

Logic Neural Networks in Board Game AI: Explore research papers and studies discussing the application of neural networks in board game AI, particularly in games like Reversi. Discuss how neural networks can be trained to predict board evaluations or strategies, potentially replacing or complementing traditional heuristic evaluation functions.

Deep Reinforcement Learning for Game Agents: Investigate the use of deep reinforcement learning techniques, such as Deep Q-Networks (DQN) or Policy Gradient methods, in training game agents for board games. Discuss their potential advantages, challenges, and performance compared to traditional Minimax-based approaches.

Feature Extraction and Representation: Discuss how machine learning techniques can assist in feature extraction and representation learning for board game states. Explore methods to identify and represent essential features of the game state for effective decision-making in Reversi.

Hybrid Approaches: Analyze research discussing hybrid approaches that combine traditional Minimax-based methods with machine learning techniques. Explore how hybrid models leverage the strengths of both approaches to enhance the performance of AI agents in board games like Reversi.

Ethical Considerations and Fairness: Discuss ethical considerations in utilizing machine learning approaches in board game AI. Address concerns related to fairness, bias, and transparency in AI decision-making, especially when using learning-based models.

CHAPTER 3

SYSTEM DESIGN

3.1 GENERAL

In this section, we would like to show how the general outline of how all the components end up working when organized and arranged together. It is further represented in the form of a flow chart below.

3.2 SYSTEM ARCHITECTURE DIAGRAM

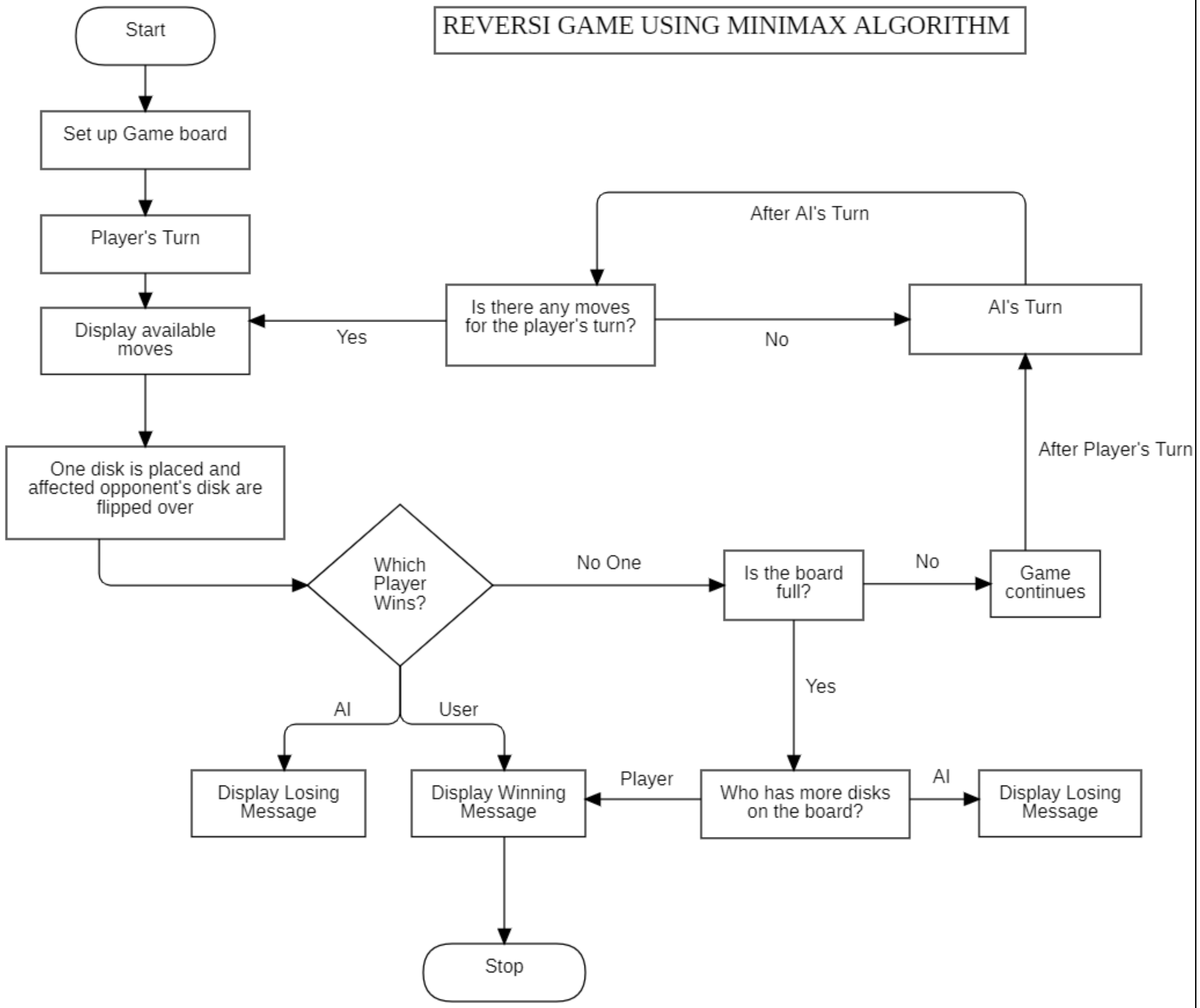

Fig 3.1: System Architecture

3.3 DEVELOPMENTAL ENVIRONMENT

3.3.1 HARDWARE REQUIREMENTS

The hardware requirements may serve as the basis for a contract for the system's implementation. It should therefore be a complete and consistent specification of the entire system. It is generally used by software engineers as the starting point for the system design.

Table 3.1 Hardware Requirements

COMPONENTS	SPECIFICATION
Computer/Device	Standard PC or Laptop
Processor	Intel Core i5
RAM	Minimum 4 GB (8 GB recommended)
Storage(Hard Disk)	Adequate for development tasks
Monitor	15" Color
Graphics Card (GPU)	Optional, as per game design
Processor Speed	Minimum 1.1 GHz
Operating System	Compatible with Python, Pygame, etc.

3.3.2 SOFTWARE REQUIREMENTS

Python: The programming language used for game development. Ensure Python is installed on your system (preferably Python 3.x).

Pygame: A Python library used for game development that facilitates graphics, sound, and user interface. Install Pygame using pip.

IDE (Integrated Development Environment): Choose an IDE for coding convenience. Popular choices include PyCharm, Visual Studio Code, or even basic text editors like Sublime Text or Atom.

3.4 DESIGN OF THE ENTIRE SYSTEM

3.4.1 ACTIVITY DIAGRAM

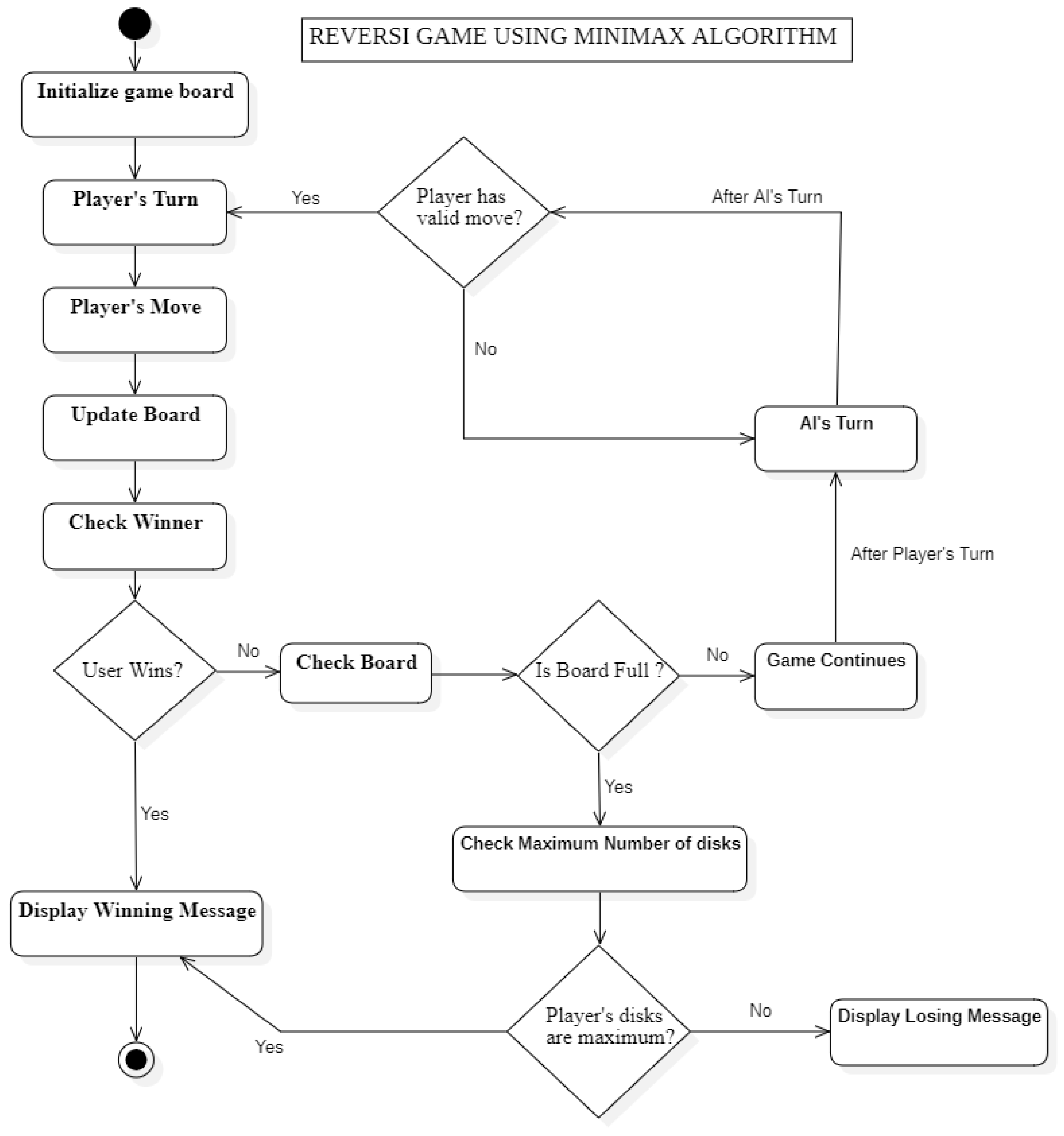

Fig 3.2 Activity Diagram

CHAPTER 4

STUDY & CONCEPTUAL DIAGRAMS

4.1 CONCEPTUAL DIAGRAM

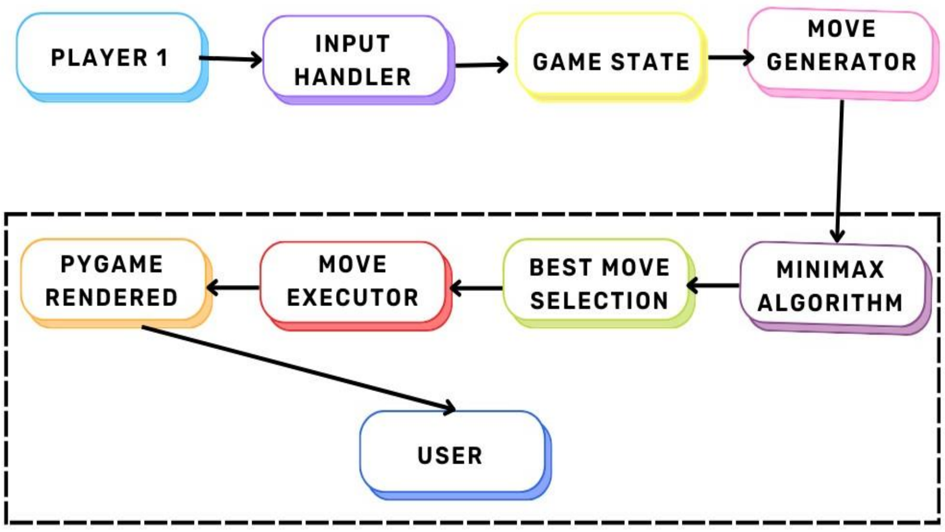

Fig 4.1: Conceptual Architecture

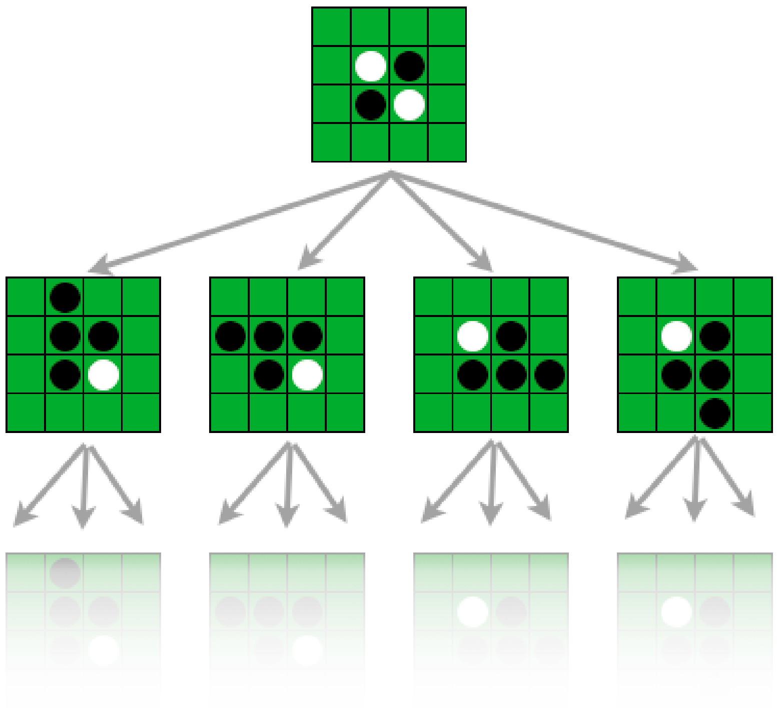

Fig 4.2: Reversi Game Tree

4.2 PROFESSIONAL VALUE OF THE STUDY

Understanding the intricacies of the minimax algorithm and its implementation in Pygame demands a grasp of algorithmic concepts and computational efficiency. Othello provides a practical context for exploring game design principles, including balancing mechanics, creating engaging gameplay, and incorporating artificial intelligence. Pygame is a popular game development library, and working with it enhances programming skills in Python, object-oriented programming, and graphics manipulation.

4.3 HIERARCHY OF DEVELOPMENT PROCESS

Problem-solving and analytical thinking: The minimax algorithm is a powerful tool for solving complex problems, and its application in Othello requires careful analysis and logical reasoning.

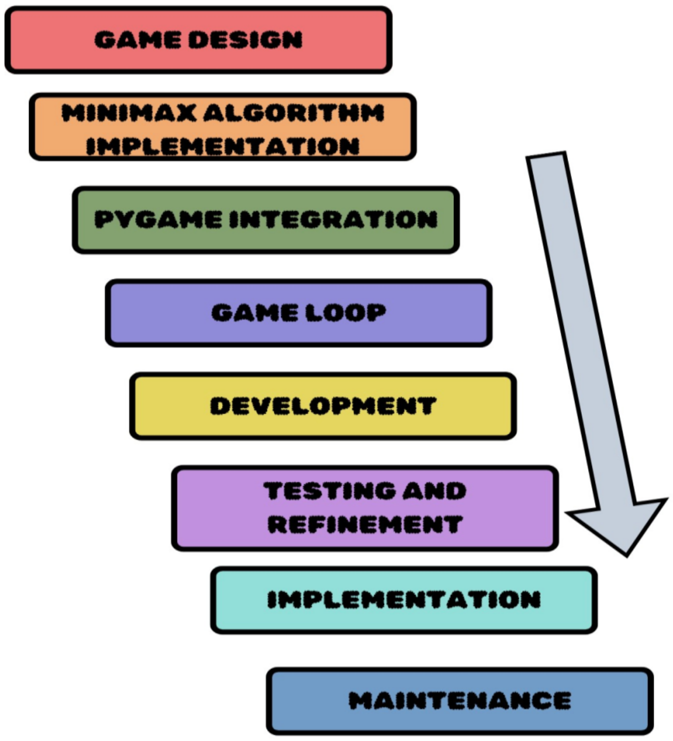

Fig 4.3: Hierarchy of development process

CHAPTER 5

PROGRAM

CODE:

```
import pygame

import random

import copy

def directions(x, y, minX=0, minY=0, maxX=7, maxY=7):

validdirections = []

if x != minX: validdirections.append((x-1, y))

if x != minX and y != minY: validdirections.append((x-1, y-1))

if x != minX and y != maxY: validdirections.append((x-1, y+1))

if x!= maxX: validdirections.append((x+1, y))

if x != maxX and y != minY: validdirections.append((x+1, y-1))

if x != maxX and y != maxY: validdirections.append((x+1, y+1))

if y != minY: validdirections.append((x, y-1))

if y != maxY: validdirections.append((x, y+1))

return validdirections

def loadImages(path, size):

img = pygame.image.load(f"{path}").convert_alpha()

img = pygame.transform.scale(img, size)

return img

def loadSpriteSheet(sheet, row, col, newSize, size):

image = pygame.Surface((32, 32)).convert_alpha()
```

```
image.blit(sheet, (0, 0), (row * size[0], col * size[1], size[0], size[1]))
image = pygame.transform.scale(image, newSize)
image.set_colorkey('Black')
return image
def evaluateBoard(grid, player):
score = 0
for y, row in enumerate(grid):
for x, col in enumerate(row):
score -= col
return score
class Othello:
def init(self):
pygame.init()
self.screen = pygame.display.set_mode((1100, 800))
pygame.display.set_caption('Othello')
self.player1 = 1
self.player2 = -1
self.currentPlayer = 1
self.time = 0
self.rows = 8
self.columns = 8
self.gameOver = False
self.gameStart=True
self.grid = Grid(self.rows, self.columns, (80, 80), self, self.screen)
self.computerPlayer = ComputerPlayer(self.grid)
self.RUN = True
def run(self):
while self.RUN == True:
self.input()
self.update()
self.draw()
def input(self):
for event in pygame.event.get():
if event.type == pygame.QUIT:
self.RUN = False
if event.type == pygame.MOUSEBUTTONDOWN:
if event.button == 3:
self.grid.printGameLogicBoard()
if event.button == 1:
if self.currentPlayer == 1 and not self.gameOver:
x, y = pygame.mouse.get_pos()
x, y = (x - 80) // 80, (y - 80) // 80
validCells = self.grid.findAvailMoves(self.grid.gridLogic, self.currentPlayer)
if not validCells: pass
else:
if (y, x) in validCells:
```

```
self.grid.insertToken(self.grid.gridLogic, self.currentPlayer, y, x)

swappableTiles = self.grid.swappableTiles(y, x, self.grid.gridLogic, self.currentPlayer)

for tile in swappableTiles:

self.grid.animateTransitions(tile, self.currentPlayer)

self.grid.gridLogic[tile[0]][tile[1]] *= -1

self.currentPlayer *= -1

self.time = pygame.time.get_ticks()

if self.gameOver:

x, y = pygame.mouse.get_pos()

if x >= 320 and x <= 480 and y >= 400 and y <= 480:

self.grid.newGame()

self.gameOver = False

def update(self):

if self.currentPlayer == -1:

new_time = pygame.time.get_ticks()

if new_time - self.time >= 100:

if not self.grid.findAvailMoves(self.grid.gridLogic, self.currentPlayer):

self.gameOver = True

return

cell, score = self.computerPlayer.computerHard(self.grid.gridLogic, 5, -64, 64, -1)

self.grid.insertToken(self.grid.gridLogic, self.currentPlayer, cell[0], cell[1])

swappableTiles = self.grid.swappableTiles(cell[0], cell[1], self.grid.gridLogic,
self.currentPlayer)

for tile in swappableTiles:
```

```
self.grid.animateTransitions(tile, self.currentPlayer)
self.grid.gridLogic[tile[0]][tile[1]] *= -1
self.currentPlayer *= -1
self.grid.player1Score = self.grid.calculatePlayerScore(self.player1)
self.grid.player2Score = self.grid.calculatePlayerScore(self.player2)
if not self.grid.findAvailMoves(self.grid.gridLogic, self.currentPlayer):
self.gameOver = True
return
def draw(self):
self.screen.fill((0, 0, 0))
self.grid.drawGrid1(self.screen)
pygame.display.update()
class Grid:
def init(self, rows, columns, size, main,screen):
self.GAME = main
self.screen=screen
self.y = rows
self.x = columns
self.size = size
self.whitetoken = loadImages('assets/WhiteToken.png', size)
self.blacktoken = loadImages('assets/BlackToken.png', size)
self.transitionWhiteToBlack = [loadImages(f'assets/BlackToWhite{i}.png', self.size) for i in range(1, 4)]
self.transitionBlackToWhite = [loadImages(f'assets/WhiteToBlack{i}.png', self.size)
for i in range(1, 4)]

self.tokens = {}

self.gridBg = self.drawGrid()

self.gridLogic = self.regenGrid(self.y, self.x)

self.player1Score = 0

self.player2Score = 0

self.font = pygame.font.SysFont('Arial', 20, True, False)

def newGame(self):

self.tokens.clear()

self.gridLogic = self.regenGrid(self.y, self.x)

def drawGrid(self):

gridBg = pygame.Surface(self.screen.get_size())

gridBg.fill((0, 0, 0))

for i in range(self.y):

for j in range(self.x):

color = (0, 128, 0)

rect = pygame.Rect(80 + j * 80, 80 + i * 80, 80, 80)

pygame.draw.rect(gridBg, color, rect)

pygame.draw.rect(gridBg, (0, 0, 0), rect, 1)

return gridBg

def regenGrid(self, rows, columns):

grid = []

for y in range(rows):
```

```
line = []
for x in range(columns):
line.append(0)
grid.append(line)
self.insertToken(grid, 1, 3, 3)
self.insertToken(grid, -1, 3, 4)
self.insertToken(grid, 1, 4, 4)
self.insertToken(grid, -1, 4, 3)
return grid
def calculatePlayerScore(self, player):
score = 0
for row in self.gridLogic:
for col in row:
if col == player:
score += 1
return score
def drawScore(self, player, score):
textImg = self.font.render(f'{player} : {score}', 1, 'White')
return textImg
def startScreen(self):if  self.GAME.gameStart:
StartScreenImg = pygame.Surface((320, 320))
startText = self.font.render('OTHELLO', 1, 'White')
StartScreenImg.blit(startText, (0, 0))
```

```
drawgrid = pygame.draw.rect(StartScreenImg, 'White', (80, 160, 160, 80))

startText = self.font.render('Start Now', 1, 'Black')

StartScreenImg.blit(startText, (120, 190))

return StartScreenImg

def endScreen(self):

if self.GAME.gameOver:

endScreenImg = pygame.Surface((320, 320))

endText = self.font.render(f'{"Congratulations, You Won!!" if self.player1Score >
self.player2Score else "Better Luck Next Time, You Lost"}', 1, 'White')

endScreenImg.blit(endText, (0, 0))

newGame = pygame.draw.rect(endScreenImg, 'White', (80, 160, 160, 80))

newGameText = self.font.render('Play Again', 1, 'Black')

endScreenImg.blit(newGameText, (120, 190))

return endScreenImg

def drawGrid1(self, window):

window.blit(self.gridBg, (0, 0))

for token in self.tokens.values():

token.draw(window)

window.blit(self.drawScore('White', self.player1Score), (900, 100))

window.blit(self.drawScore('Black', self.player2Score), (900, 200))

availMoves = self.findAvailMoves(self.gridLogic, self.GAME.currentPlayer)

if self.GAME.currentPlayer == 1:

for move in availMoves:
```

```
pygame.draw.rect(window, 'White', (80 + (move[1] * 80) + 30, 80 + (move[0] * 80) +
30, 20, 20))

if self.GAME.gameOver:

window.blit(self.endScreen(), (240, 240))

def printGameLogicBoard(self):

print('  | A | B | C | D | E | F | G | H |')

for i, row in enumerate(self.gridLogic):

line = f'{i} |'.ljust(3, " ")

for item in row:

line += f"{item}".center(3, " ") + '|'

print(line)

print()

def findValidCells(self, grid, curPlayer):

validCellToClick = []

for gridX, row in enumerate(grid):

for gridY, col in enumerate(row):

if grid[gridX][gridY] != 0:

continue

DIRECTIONS = directions(gridX, gridY)

for direction in DIRECTIONS:

dirX, dirY = direction

checkedCell = grid[dirX][dirY]

if checkedCell == 0 or checkedCell == curPlayer:

continue
```

```
if (gridX, gridY) in validCellToClick:
continue
validCellToClick.append((gridX, gridY))
return validCellToClick
def swappableTiles(self, x, y, grid, player):
surroundCells = directions(x, y)
if len(surroundCells) == 0:
return []
swappableTiles = []
for checkCell in surroundCells:
checkX, checkY = checkCell
difX, difY = checkX - x, checkY - y
currentLine = []
RUN = True
while RUN:
if grid[checkX][checkY] == player * -1:
currentLine.append((checkX, checkY))
elif grid[checkX][checkY] == player:
RUN = False
break
elif grid[checkX][checkY] == 0:
currentLine.clear()
RUN = False
```

```
checkX += difX
checkY += difY
if checkX < 0 or checkX > 7 or checkY < 0 or checkY > 7:
currentLine.clear()
RUN = False
if len(currentLine) > 0:
swappableTiles.extend(currentLine)
return swappableTiles
def findAvailMoves(self, grid, currentPlayer):
validCells = self.findValidCells(grid, currentPlayer)
playableCells = []
for cell in validCells:
x, y = cell
if cell in playableCells:
continue
swapTiles = self.swappableTiles(x, y, grid, currentPlayer)
if len(swapTiles) > 0:
playableCells.append(cell)
return playableCells
def insertToken(self, grid, curplayer, y, x):
tokenImage = self.whitetoken if curplayer == 1 else self.blacktoken
self.tokens[(y, x)] = Token(curplayer, y, x, tokenImage, self.GAME)
grid[y][x] = self.tokens[(y, x)].player
```

```
def animateTransitions(self, cell, player):
if player == 1:
self.tokens[(cell[0], cell[1])].transition(self.transitionWhiteToBlack, self.whitetoken)
else:
self.tokens[(cell[0], cell[1])].transition(self.transitionBlackToWhite, self.blacktoken)
class Token:
def init(self, player, gridX, gridY, image, main):
self.player = player
self.gridX = gridX
self.gridY = gridY
self.posX = 80 + (gridY * 80)
self.posY = 80 + (gridX * 80)
self.GAME = main
self.image = image
def transition(self, transitionImages, tokenImage):
for i in range(30):
self.image = transitionImages[i // 10]
self.GAME.draw()
self.image = tokenImage
def draw(self, window):
window.blit(self.image, (self.posX, self.posY))
class ComputerPlayer:
def init(self, gridObject):
```

```
self.grid = gridObject
def computerHard(self, grid, depth, alpha, beta, player):
newGrid = copy.deepcopy(grid)
availMoves = self.grid.findAvailMoves(newGrid, player)
if depth == 0 or len(availMoves) == 0:
bestMove, Score = None, evaluateBoard(grid, player)
return bestMove, Score
if player < 0:
bestScore = -64
bestMove = None
for move in availMoves:
x, y = move
swappableTiles = self.grid.swappableTiles(x, y, newGrid, player)
newGrid[x][y] = player
for tile in swappableTiles:
newGrid[tile[0]][tile[1]] = player
bMove, value = self.computerHard(newGrid, depth-1, alpha, beta, player *-1)
if value > bestScore:
bestScore = value
bestMove = move
alpha = max(alpha, bestScore)
if beta <= alpha: break
newGrid = copy.deepcopy(grid)
```

```
return bestMove, bestScore
if player > 0:
bestScore = 64
bestMove = None
for move in availMoves:
x, y = move
swappableTiles = self.grid.swappableTiles(x, y, newGrid, player)
newGrid[x][y] = player
for tile in swappableTiles:
newGrid[tile[0]][tile[1]] = player
bMove, value = self.computerHard(newGrid, depth-1, alpha, beta, player)
if value < bestScore:
bestScore = value
bestMove = move
beta = min(beta, bestScore)
if beta <= alpha: break
newGrid = copy.deepcopy(grid)
return bestMove, bestScore
if name == 'main':
game = Othello()
game.run()
pygame.quit()
```

CHAPTER 6

RESULTS AND DISCUSSIONS

6.1 OUTPUT

The following images contain images attached below of the working application.

The following image in Fig 5.1 shows the game play between user and the computer in the Application.

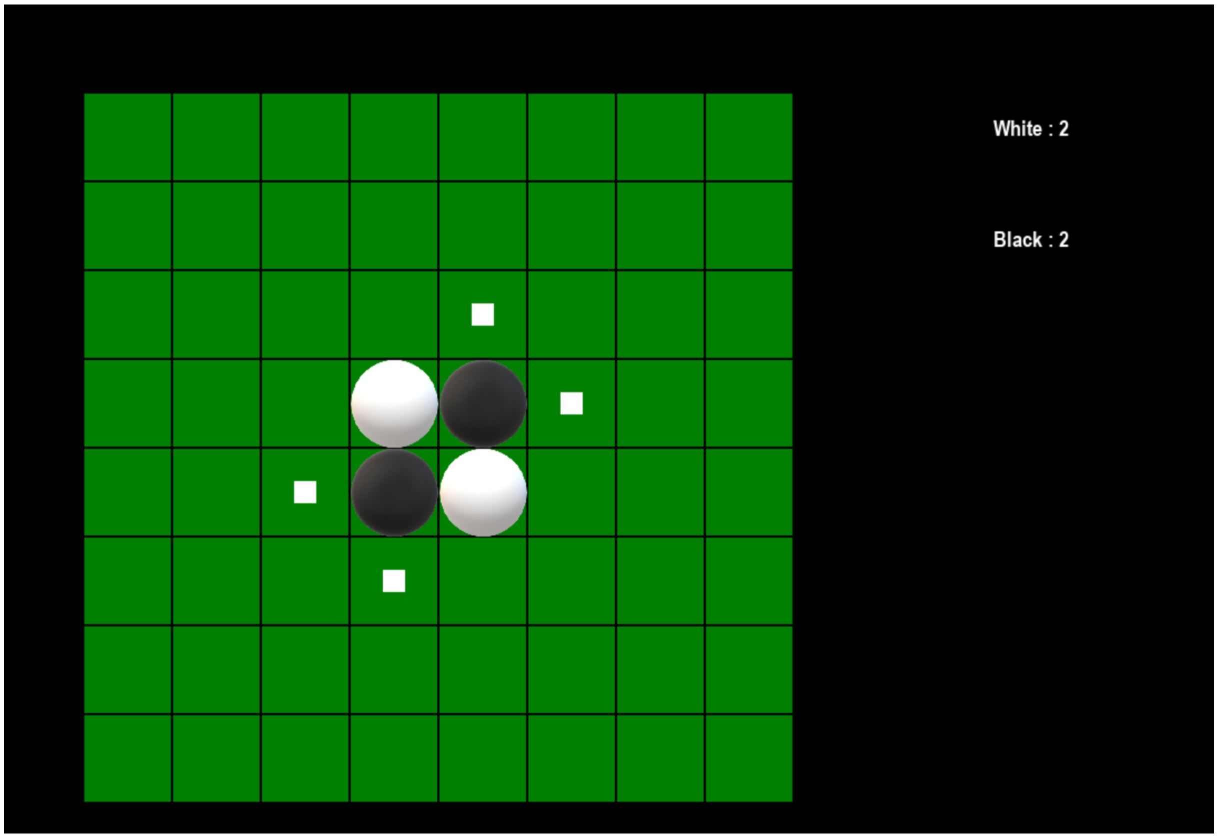

Fig 6.1: Output

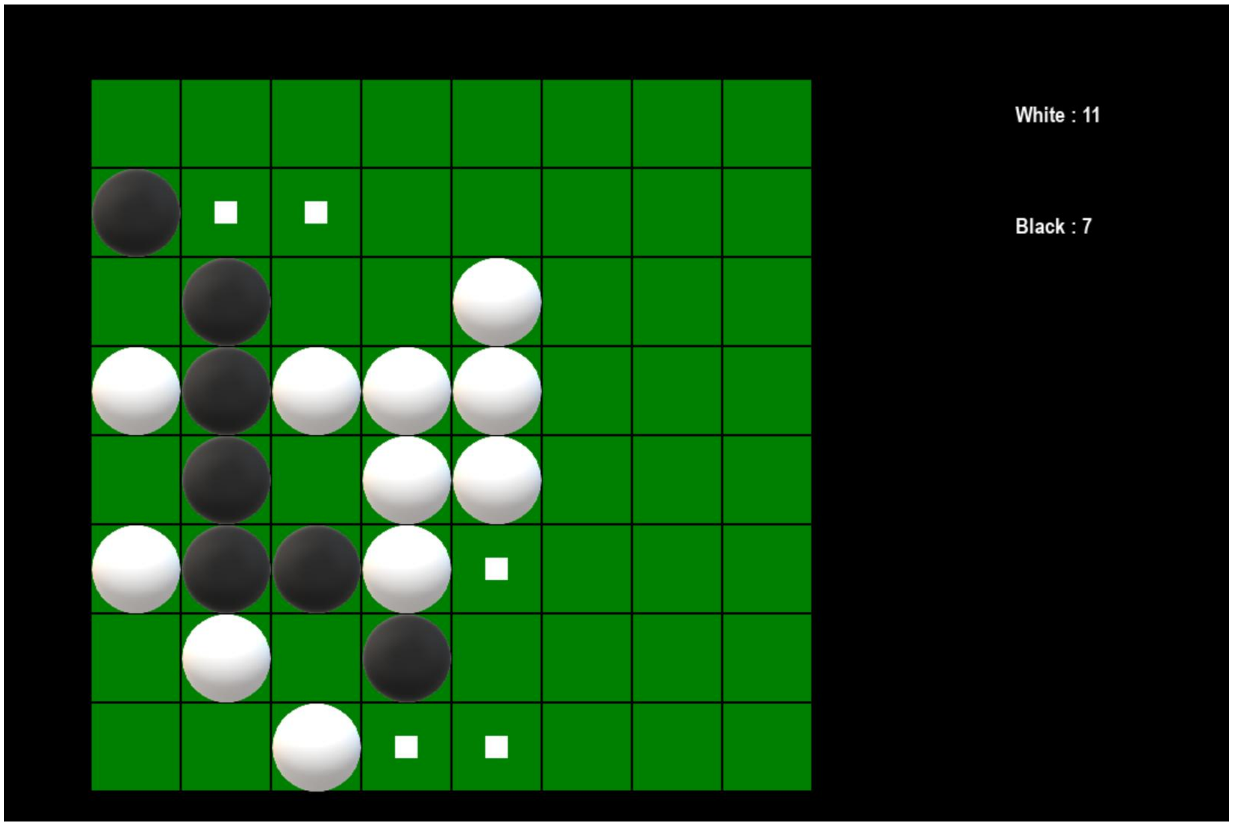

Fig 6.2: The player starts to play the game

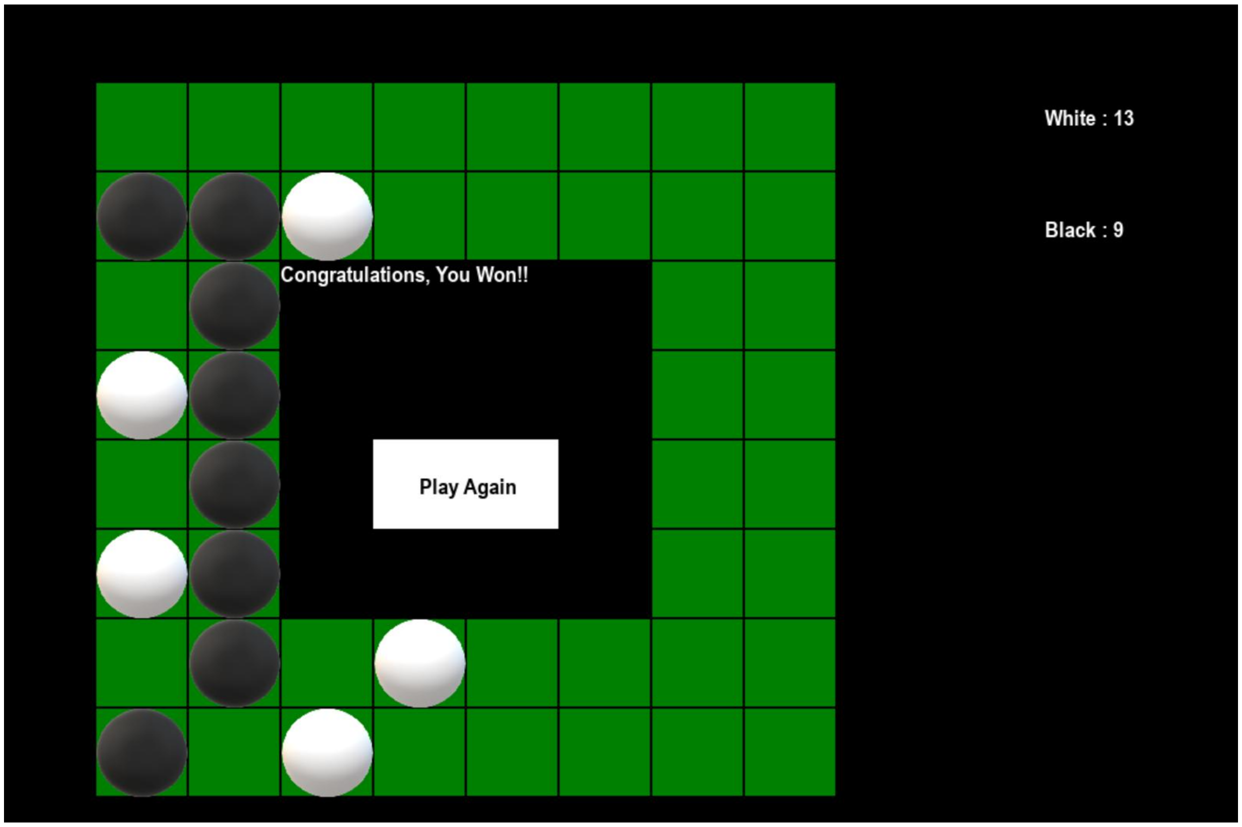

Fig 6.3: The player wins the game

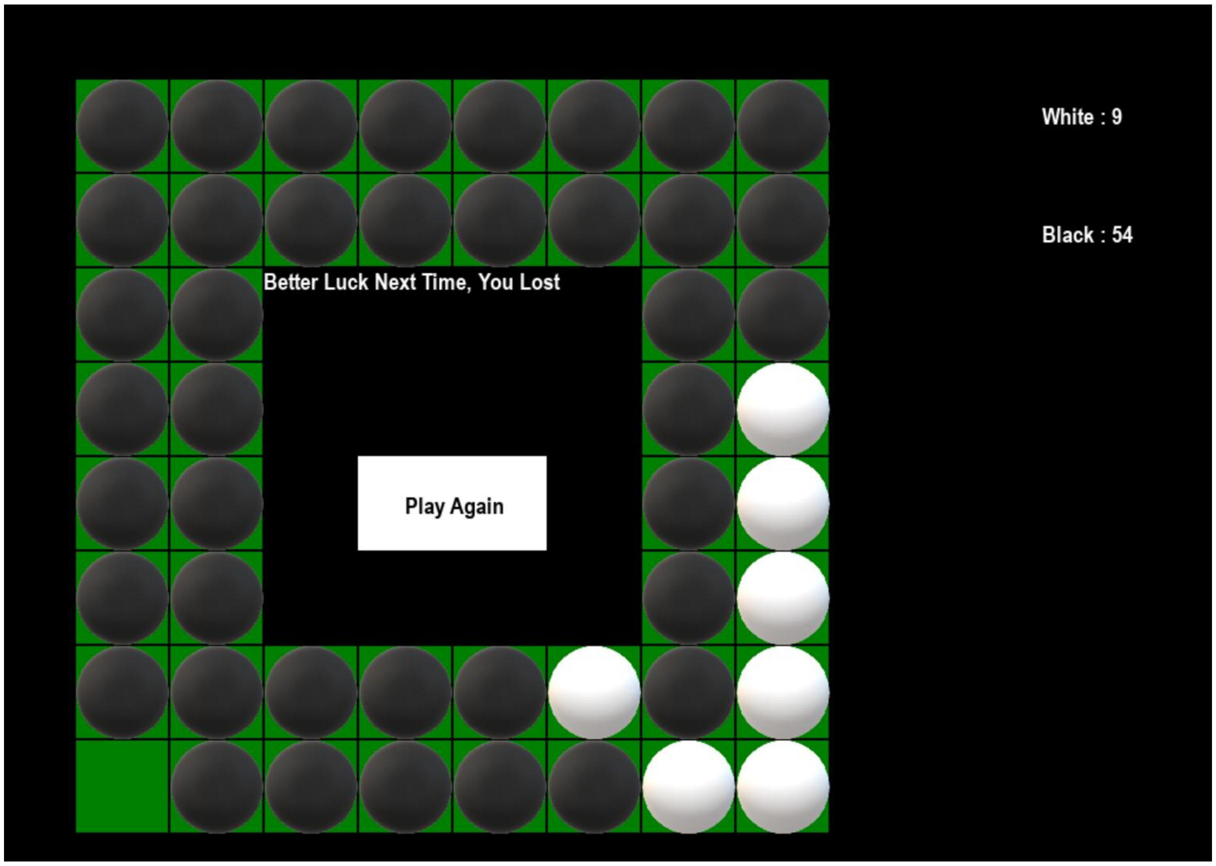

Fig 6.4: The player lose the game

6.2 RESULT

The minimax algorithm provides a structured approach to evaluating game states and making informed decisions, leading to a more strategic and challenging AI opponent. The combination of Pygame and the minimax algorithm facilitates a hands-on approach to game development, allowing players to experiment with different AI strategies and observe their impact on gameplay. The integration of minimax with Pygame in the Othello game offers a valuable platform for exploring artificial intelligence, game design, and problem-solving, while providing an engaging and challenging gaming experience for players

CHAPTER 7

CONCLUSION AND FUTURE ENHANCEMENT

7.1 CONCLUSION

In conclusion, the provided Python code successfully implements the Othello (Reversi) game, combining game logic, graphical user interface, and an artificial intelligence (AI) opponent. The code is built using the Pygame library, allowing for a visually appealing and interactive gaming experience. The game includes features such as player input recognition, token animations, and an AI opponent that utilizes the minimax algorithm for strategic decision-making.

The Othello game demonstrates the core principles of the classic board game, where players aim to dominate the board by strategically placing their tokens and flipping their opponent's tokens. The minimax algorithm employed in the AI player ensures a competitive opponent by considering future game states and selecting moves that lead to the best possible outcome.

7.2 FUTURE ENHANCEMENT

- Enhancement of user interface. This could involve smoother transitions for token movements and more visually appealing game elements.
- Introduce multiplayer functionality, enabling users to play against each other.
- Track and display game statistics, such as the number of games played, win or loss ratios, and average game duration.

REFERENCES

[1] https://www.instructables.com/Othello-Artificial-Intelligence/

[2] https://taylorkemp.github.io/jekyll/update/2018/08/24/welcome-to-jekyll.html

[3] pygame v2.6.0 documentation : https://www.pygame.org/docs/ref/pygame.html

[4] https://www.worldothello.org/about/about-othello/othello-rules/official-rules/english

[5] Engel, K. T. Learning a Reversi Board Evaluator with Minimax.

[6] Strong, G. (2011). The minimax algorithm. Trinity College Dublin.

[7] Lu, K. (2014). The Game Theory of Reversi.

[8] S. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, Prentice Hall, Third Edition, 2015.

[9] Nils J. Nilsson, Artificial Intelligence: A New Synthesis (1 ed.), Morgan-Kaufmann, 1998. ISBN 978- 1558605350.

[10] https://medium.com/@gmu1233/how-to-write-an-othello-ai-with-alpha-beta-search-58131ffe67eb

[11] https://u.osu.edu/fe1181au18sec23554a/software-documentation/final-pseudocode/