PROJECT REPORT

Introduction: -

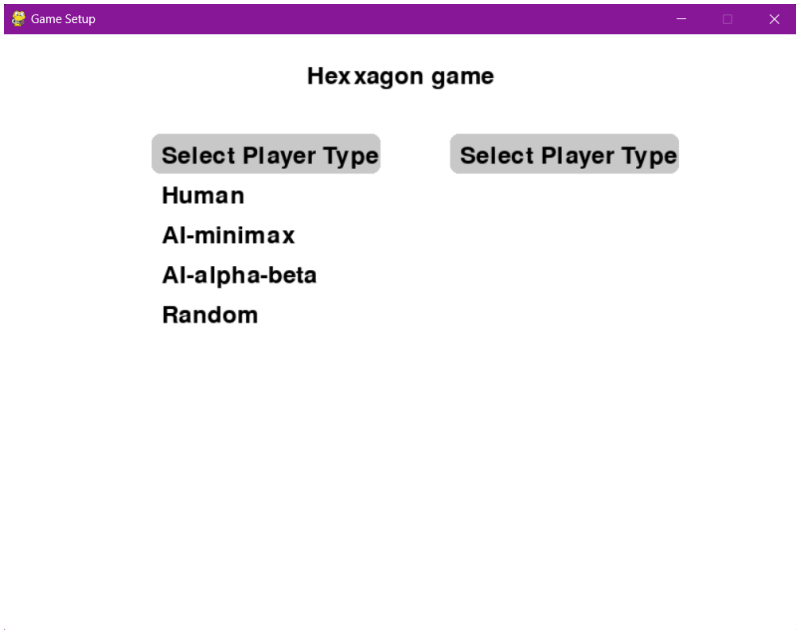
This project involves the development of a strategic board game called *Hexxagon* using Python and the pygame library. The game supports multiple modes, including Human vs Human, AI vs AI, and Human vs AI, with AI strategies implemented using Minimax and Alpha-Beta pruning algorithms. The objective is to provide an engaging gameplay experience while showcasing the implementation of advanced AI techniques.

Objectives

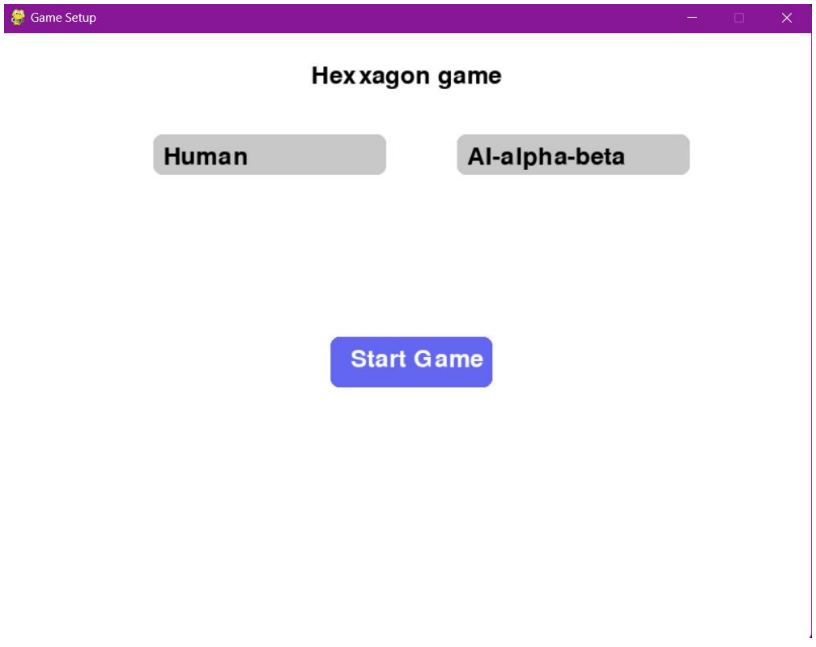
1. **Create a visually appealing hexagonal grid-based board game.**
2. **Implement various player types:**
    - **Human players.**
    - **AI players using Minimax and Alpha-Beta pruning.**
    - **Random move generators.**
3. **Provide multiple gameplay modes:**
    - **Human vs Human.**
    - **Human vs AI (minimax).**
    - **Human vs AI (alpha-beta).**
    - **Human vs Random**
    - **AI (minimax) vs AI (minimax).**
    - **AI (minimax) vs AI (alpha-beta).**
    - **AI (minimax) vs Random.**
    - **AI (alpha-beta) vs AI (alpha-beta).**
    - **AI (alpha-beta) vs Random.**
    - **Random vs Random.**
4. **Ensure smooth interaction and user-friendly UI.**

Results Screenshots: -

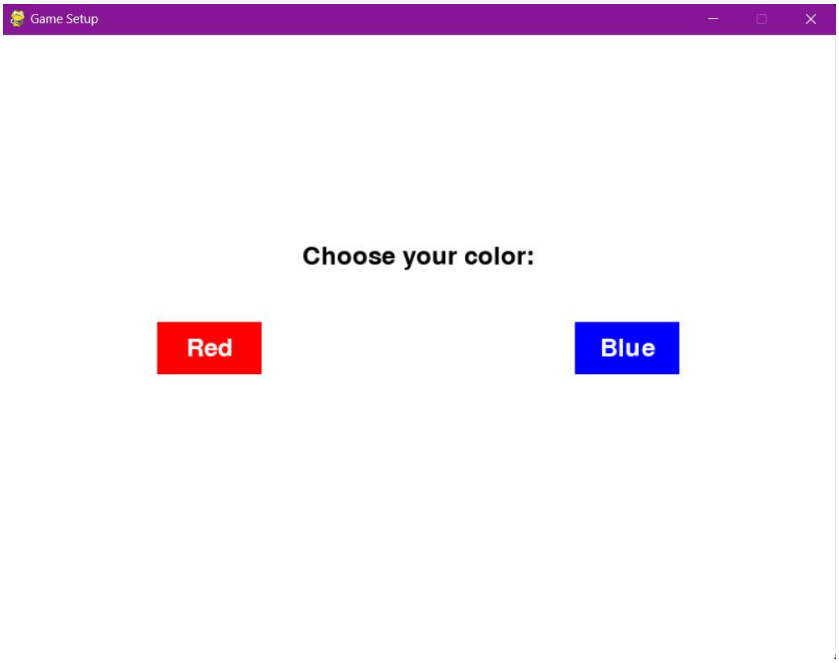
1.Main Menu



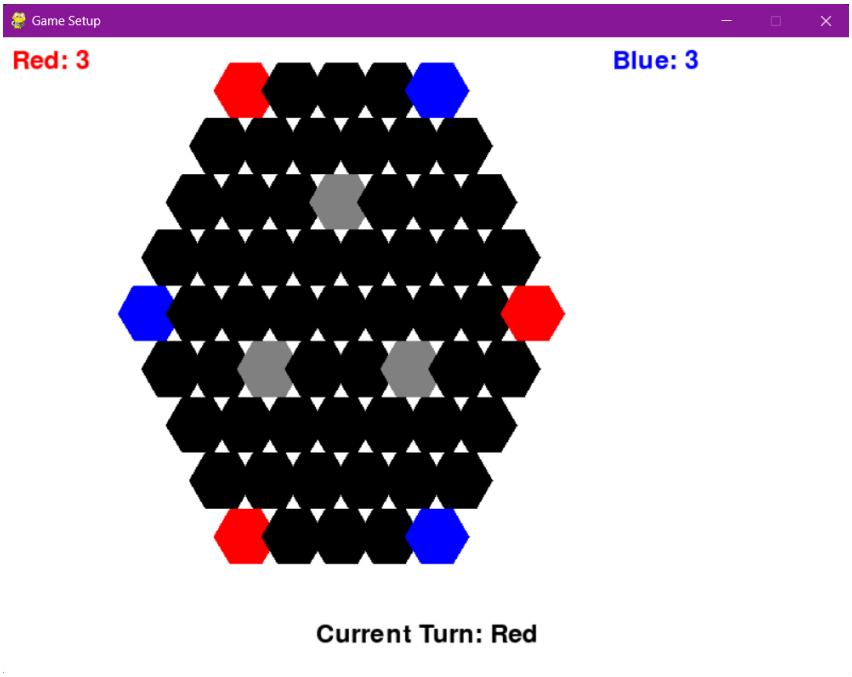
2.Start Game



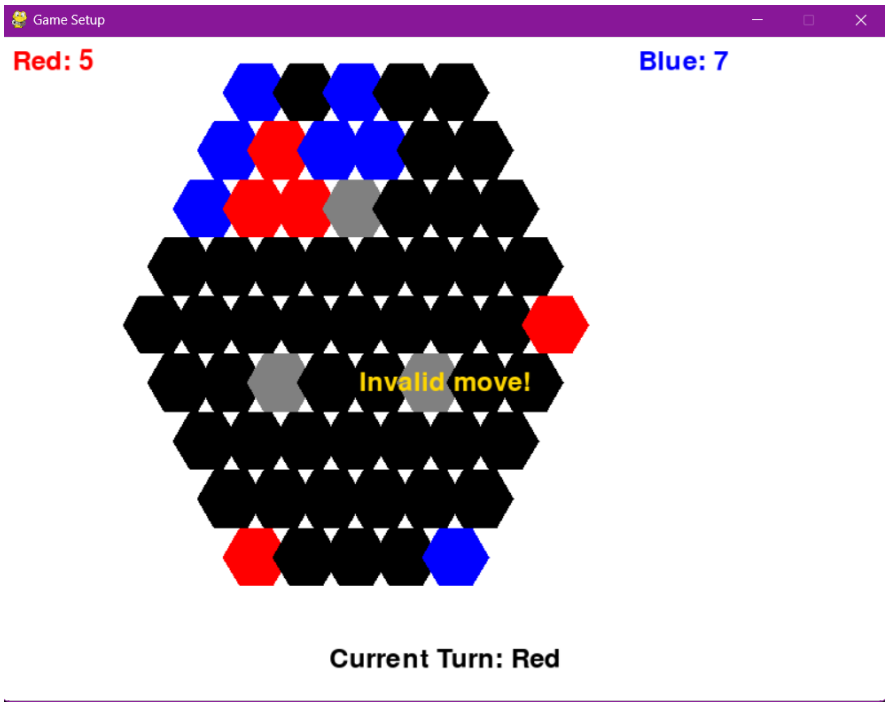
3.Choosing Color



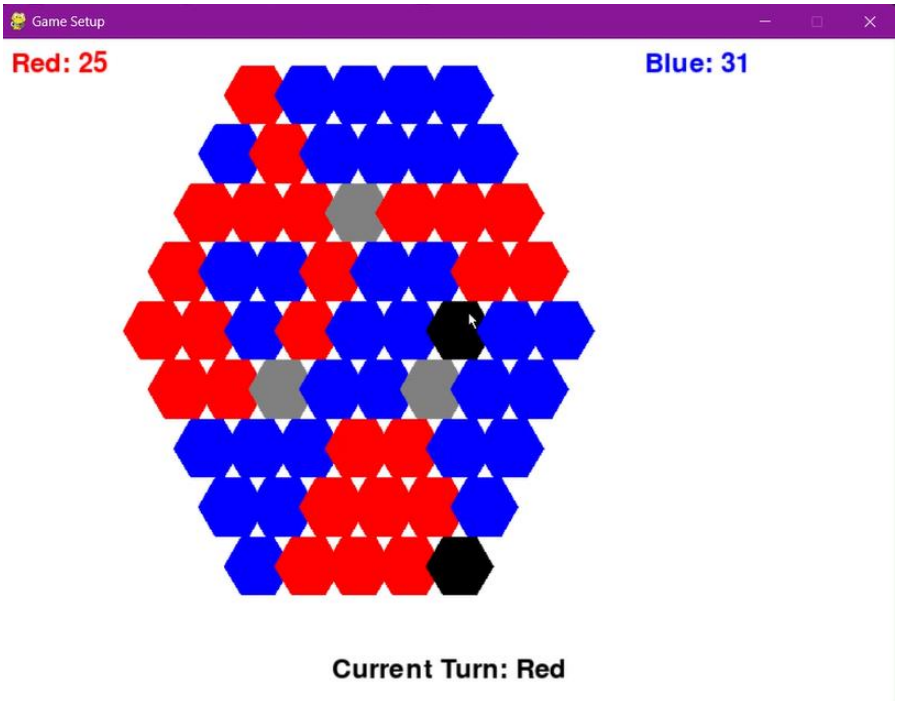
4.Initial setup



5.Invalid Move



6.Winning moment



7.Winner Declaration

Code Screenshots: -

1.All modules

```python
#required modules
import pygame
import sys
import math
import copy
import random
```

✓ 0.4s Python

```
pygame 2.6.1 (SDL 2.28.4, Python 3.10.0)
Hello from the pygame community. https://www.pygame.org/contribute.html
```

2.Setting up pygame

```python
#initialize pygame
pygame.init()

# Constants
WIDTH, HEIGHT = 800, 600
WHITE = (255, 255, 255)
GRAY = (200, 200, 200)
BLACK = (0, 0, 0)
PRIMARY = (99, 102, 241)
SECONDARY = (79, 70, 229)
FONT = pygame.font.Font(None, 36)

# Setup screen
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Game Setup")

# Options
player_options = ["Human", "AI-minimax", "AI-alpha-beta", "Random"]
player1_selection = None
player2_selection = None
```

```python
# Button class
class Button:
    def __init__(self, x, y, width, height, text, callback):
        self.rect = pygame.Rect(x, y, width, height)
        self.text = text
        self.callback = callback
        self.hovered = False

    def draw(self, screen):
        color = SECONDARY if self.hovered else PRIMARY
        pygame.draw.rect(screen, color, self.rect, border_radius=8)
        text_surface = FONT.render(self.text, True, WHITE)
        screen.blit(text_surface, (self.rect.x + 20, self.rect.y + 10))

    def check_event(self, event):
        if event.type == pygame.MOUSEMOTION:
            self.hovered = self.rect.collidepoint(event.pos)
        if event.type == pygame.MOUSEBUTTONDOWN and self.rect.collidepoint(event.pos):
            self.callback()
```

```python
# Dropdown class
class Dropdown:
    def __init__(self, x, y, width, height, options, callback):
        self.rect = pygame.Rect(x, y, width, height)
        self.options = options
        self.callback = callback
        self.expanded = False
        self.selected = "Select Player Type"

    def draw(self, screen):
        pygame.draw.rect(screen, GRAY, self.rect, border_radius=8)
        text_surface = FONT.render(self.selected, True, BLACK)
        screen.blit(text_surface, (self.rect.x + 10, self.rect.y + 10))

        if self.expanded:
            for i, option in enumerate(self.options):
                option_rect = pygame.Rect(self.rect.x, self.rect.y + (i + 1) * 40, self.rect.width, self.rect.height)
                pygame.draw.rect(screen, WHITE, option_rect, border_radius=8)
                option_text = FONT.render(option, True, BLACK)
                screen.blit(option_text, (option_rect.x + 10, option_rect.y + 10))

    def check_event(self, event):
        if event.type == pygame.MOUSEBUTTONDOWN:
            if self.rect.collidepoint(event.pos):
                self.expanded = not self.expanded
            elif self.expanded:
                for i, option in enumerate(self.options):
                    option_rect = pygame.Rect(self.rect.x, self.rect.y + (i + 1) * 40, self.rect.width, self.rect.height)
                    if option_rect.collidepoint(event.pos):
                        self.selected = option
                        self.expanded = False
                        self.callback(option)
                        break
                else:
                    self.expanded = False

# Callbacks
def select_player1(option):
    global player1_selection
    player1_selection = option

def select_player2(option):
    global player2_selection
    player2_selection = option
```

✓ 0.7s Python

3.Utility Functions

```python
WIDTH, HEIGHT = 800, 600
WHITE = (255, 255, 255)
RED = (255, 0, 0)
BLUE = (0, 0, 255)
BLACK = (0, 0, 0)
HEX_SIZE = 30

pygame.font.init()
font = pygame.font.Font(None, 36)

def display_winner(screen, winner_text):
    screen.fill(WHITE)
    large_font = pygame.font.Font(None, 72)
    text = large_font.render(winner_text, True, BLACK)
    text_rect = text.get_rect(center=(WIDTH//2, HEIGHT//2))
    screen.blit(text, text_rect)
    pygame.display.flip()

def calculate_scores(grid):
    """Calculate scores for Red and Blue players."""
    red_score = sum(row.count("R") for row in grid)
    blue_score = sum(row.count("B") for row in grid)
    return red_score, blue_score
```

```python
def display_scores(screen, red_score, blue_score):
    """Display scores on the screen."""
    red_text = font.render(f"Red: {red_score}", True, RED)
    blue_text = font.render(f"Blue: {blue_score}", True, BLUE)

    # Positioning
    screen.blit(red_text, (10, 10))  # Top-left corner
    screen.blit(blue_text, (WIDTH - 225, 10))  # Top-right corner

# Function to draw a hexagon
def draw_hexagon(x, y, color):
    points = []
    for i in range(6):
        angle = math.pi / 3 * i
        px = x + math.cos(angle) * HEX_SIZE
        py = y + math.sin(angle) * HEX_SIZE
        points.append((px, py))
    pygame.draw.polygon(screen, color, points)

# Function to draw the grid
def draw_grid(grid):
    screen.fill(WHITE)
    for row, cols in enumerate(grid):
        offset = (row % 2) * HEX_SIZE * 0.75
        for col, cell in enumerate(cols):
            x = col * HEX_SIZE * 1.5 + offset + 50
            y = row * HEX_SIZE * 1.75 + 50
            if cell == "R":
                draw_hexagon(x, y, RED)
            elif cell == "B":
                    draw_hexagon(x, y, BLUE)
            elif cell == "●":
                    draw_hexagon(x, y, BLACK)
            elif cell == "O":
                    draw_hexagon(x, y, (128, 128, 128))  # Gray for blocked hex

def get_neighbors(row, col):
    if row%2 == 0:
        return [
            (row, col - 1), (row, col + 1), (row - 1, col), (row - 1, col - 1),
            (row + 1, col), (row + 1, col - 1)
        ]
    else:
        return [
            (row, col - 1), (row, col + 1), (row - 1, col), (row - 1, col + 1),
            (row + 1, col), (row + 1, col + 1)
        ]
def get_jump_positions(row, col):
    if row%2 == 0:
        return [
            (row, col - 2), (row, col + 2), (row - 1, col - 2), (row - 1, col + 1),
            (row - 2, col), (row - 2, col + 1), (row - 2, col - 1),
            (row + 1, col - 2), (row + 1, col + 1), (row + 2, col),
            (row + 2, col + 1), (row + 2, col - 1)
        ]
    else:
        return [
            (row, col - 2), (row, col + 2), (row - 1, col - 1), (row - 1, col + 2),
            (row - 2, col), (row - 2, col + 1), (row - 2, col - 1),
            (row + 1, col - 1), (row + 1, col + 2), (row + 2, col),
            (row + 2, col + 1), (row + 2, col - 1)
        ]

def is_grid_full(grid):
    """Check if the grid is full."""
    for row in grid:
        if "●" in row:  # Empty hexagon
            return False
    return True

def declare_winner(grid):
    """Declare the winner based on the number of pieces."""
    red_count = sum(row.count("R") for row in grid)
    blue_count = sum(row.count("B") for row in grid)
    if red_count > blue_count:
        return "Red wins!"
    elif blue_count > red_count:
        return "Blue wins!"
    else:
        return "It's a tie!"

def get_valid_moves(grid, player):
    """Find all valid moves (clone or jump) for the given player."""
    moves = []
    for r in range(len(grid)):
        for c in range(len(grid[r])):
            if grid[r][c] == player:
                for nr, nc in get_neighbors(r, c) + get_jump_positions(r, c):
                    if 0 <= nr < len(grid) and 0 <= nc < len(grid[nr]) and grid[nr][nc] == "●":
                        moves.append((r, c, nr, nc))
    return moves

def set_hexagon(grid, row, col, symbol):
    if 0 <= row < len(grid) and 0 <= col < len(grid[row]):
        grid[row][col] = symbol

def evaluate_board(grid, player):
    """Heuristic function to evaluate board state."""
    opponent = 'R' if player == 'B' else 'B'
    player_score = sum(row.count(player) for row in grid)
    opponent_score = sum(row.count(opponent) for row in grid)
    return player_score - opponent_score
```

```python
def execute_move(grid, move, player):
    """Execute the move (clone or jump) on the board."""
    r, c, nr, nc = move
    if grid[nr][nc] == "O":
        return  # Cannot move into blocked hex
    grid[nr][nc] = player
    if (nr,nc) in get_jump_positions(r,c):
        grid[r][c] = '●'  # Jump move leaves previous spot empty

    # Change adjacent opponent stones
    opponent = 'R' if player == 'B' else 'B'
    for nr, nc in get_neighbors(nr, nc):
        if 0 <= nr < len(grid) and 0 <= nc < len(grid[nr]) and grid[nr][nc] == opponent:
            grid[nr][nc] = player

def create_hexagonal_grid(size=4):
    hex_grid = []

    # Create hexagonal grid structure
    for row in range(size):
        cols = size + row
        hex_grid.append(["."] * cols)
    for row in range(size - 2, -1, -1):
        cols = size + row
        hex_grid.append(["."] * cols)

    return hex_grid
def check_no_valid_moves(grid, current_player):
    """Check if current player has no valid moves and let opponent fill remaining spaces."""
    opponent = 'R' if current_player == 'B' else 'B'

    # Check if current player has moves
    if not get_valid_moves(grid, current_player):
        # Opponent fills all empty spaces
        for row in range(len(grid)):
            for col in range(len(grid[row])):
                if grid[row][col] == '●':
                    grid[row][col] = opponent
        return True
    return False

def display_turn(screen, current_player):
    """Display the current player's turn on the screen."""
    turn_text = FONT.render(f"Current Turn: {'Red' if current_player == 'R' else 'Blue'}", True, BLACK)
    screen.blit(turn_text, (WIDTH // 2 - turn_text.get_width() // 2, HEIGHT - 50))  # Position at the bottom center
def set_hexgrid():
    # Create and display the hexagonal grid
    hex_grid = create_hexagonal_grid(10)

    for row, start, end in [
        (0, 4, 8),
        (1, 3, 8),
        (2, 3, 9),
        (3, 2, 9),
        (4, 2, 10),
        (5, 2, 9),
        (6, 3, 9),
        (7, 3, 8),
        (8, 4, 8)
    ]:
        for col in range(start, end + 1):
            set_hexagon(hex_grid, row, col, "●")

    set_hexagon(hex_grid, 0, 4, "R")
    set_hexagon(hex_grid, 0, 8, "B")
    set_hexagon(hex_grid, 4, 2, "B")
    set_hexagon(hex_grid, 4, 10,"R")
    set_hexagon(hex_grid, 8, 4,"R")
    set_hexagon(hex_grid, 8, 8,"B")
    set_hexagon(hex_grid,2,6,"O")
    set_hexagon(hex_grid,5,4,"O")
    set_hexagon(hex_grid,5,7,"O")

    return hex_grid
def select_color():
    """Prompt the human player to select their color."""
    pygame.init()
    screen = pygame.display.set_mode((WIDTH, HEIGHT))
    font = pygame.font.Font(None, 36)
    clock = pygame.time.Clock()

    # Button positions
    red_button = pygame.Rect(WIDTH // 4 - 50, HEIGHT // 2 - 25, 100, 50)
    blue_button = pygame.Rect(3 * WIDTH // 4 - 50, HEIGHT // 2 - 25, 100, 50)

    selected_color = None

    while selected_color is None:
        screen.fill(WHITE)

        # Display instructions
        text = font.render("Choose your color:", True, BLACK)
        screen.blit(text, (WIDTH // 2 - text.get_width() // 2, HEIGHT // 3))

        # Draw buttons
        pygame.draw.rect(screen, RED, red_button)
        pygame.draw.rect(screen, BLUE, blue_button)
```

```python
        # Button labels
        red_text = font.render("Red", True, WHITE)
        blue_text = font.render("Blue", True, WHITE)
        screen.blit(red_text, (red_button.x + red_button.width // 2 - red_text.get_width() // 2,
                               red_button.y + red_button.height // 2 - red_text.get_height() // 2))
        screen.blit(blue_text, (blue_button.x + blue_button.width // 2 - blue_text.get_width() // 2,
                                blue_button.y + blue_button.height // 2 - blue_text.get_height() // 2))

        pygame.display.flip()

        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                sys.exit()
            elif event.type == pygame.MOUSEBUTTONDOWN:
                if red_button.collidepoint(event.pos):
                    selected_color = "R"
                elif blue_button.collidepoint(event.pos):
                    selected_color = "B"

        clock.tick(30)

    return selected_color
```

0.0s Python

4. AI-Algorithms

```python
def minimax(grid, depth, maximizing, player):
    """Minimax algorithm with depth limit."""
    if depth == 0:
        return evaluate_board(grid, player), None

    best_move = None
    valid_moves = get_valid_moves(grid, player)
    opponent = 'R' if player == 'B' else 'B'

    if maximizing:
        max_eval = float('-inf')
        for move in valid_moves:
            new_grid = copy.deepcopy(grid)
            execute_move(new_grid, move, player)
            eval_score, _ = minimax(new_grid, depth - 1, False, opponent)
            if eval_score > max_eval:
                max_eval, best_move = eval_score, move
        return max_eval, best_move
    else:
        min_eval = float('inf')
        for move in valid_moves:
            new_grid = copy.deepcopy(grid)
            execute_move(new_grid, move, opponent)
            eval_score, _ = minimax(new_grid, depth - 1, True, player)
            if eval_score < min_eval:
                min_eval, best_move = eval_score, move
        return min_eval, best_move
def ai_play_minmax(grid, player):
    """AI selects the best move using Minimax and executes it."""
    _, best_move = minimax(grid, depth=3, maximizing=True, player=player)
    if best_move:
        r, c, nr, nc = best_move
        move_type = "jump" if (nr, nc) in get_jump_positions(r, c) else "move"
        execute_move(grid, best_move, player)
        print(f"AI ({player}) performs a {move_type} from {best_move[:2]} to {best_move[2:]}")
    else:
        print("AI has no valid moves!")
def minimax_alpha_beta(grid, depth, alpha, beta, maximizing, player):
    if depth == 0:
        return evaluate_board(grid, player), None

    best_move = None
    valid_moves = get_valid_moves(grid, player)
    opponent = 'R' if player == 'B' else 'B'

    if maximizing:
        max_eval = float('-inf')
        for move in valid_moves:
            new_grid = copy.deepcopy(grid)
            execute_move(new_grid, move, player)
            eval_score, _ = minimax_alpha_beta(new_grid, depth - 1, alpha, beta, False, opponent)
            if eval_score > max_eval:
                max_eval, best_move = eval_score, move
            alpha = max(alpha, max_eval)
            if beta <= alpha:
                break  # Beta cut-off
        return max_eval, best_move
    else:
        min_eval = float('inf')
        for move in valid_moves:
            new_grid = copy.deepcopy(grid)
            execute_move(new_grid, move, opponent)
            eval_score, _ = minimax_alpha_beta(new_grid, depth - 1, alpha, beta, True, player)
            if eval_score < min_eval:
                min_eval, best_move = eval_score, move
            beta = min(beta, min_eval)
            if beta <= alpha:
                break  # Alpha cut-off
        return min_eval, best_move
```

```python
def ai_play_alphabeta(grid, player):
    """AI selects the best move using Minimax with alpha-beta pruning and executes it."""
    _, best_move = minimax_alpha_beta(grid, depth=3, alpha=float('-inf'), beta=float('inf'), maximizing=True, player=player)
    if best_move:
        r, c, nr, nc = best_move
        move_type = "jump" if (nr, nc) in get_jump_positions(r, c) else "move"
        execute_move(grid, best_move, player)
        print(f"AI ({player}) performs a {move_type} from {best_move[:2]} to {best_move[2:]}")
    else:
        print("AI has no valid moves!")

def random_play(grid, player):
        """Randomly selects a move from available valid moves and executes it."""
        valid_moves = get_valid_moves(grid, player)
        if valid_moves:
            move = random.choice(valid_moves)
            r, c, nr, nc = move
            move_type = "jump" if (nr, nc) in get_jump_positions(r, c) else "move"
            execute_move(grid, move, player)
            print(f"Player ({player}) performs a {move_type} from {move[:2]} to {move[2:]}")
        else:
            print(f"{player} has no valid moves!")
```

5.Implementation of Gaming modes

- **Human vs Human.**
- **Human vs AI (minimax).**
- **Human vs AI (alpha-beta).**
- **Human vs Random**
- **AI (minimax) vs AI (minimax).**
- **AI (minimax) vs AI (alpha-beta).**
- **AI (minimax) vs Random.**
- **AI (alpha-beta) vs AI (alpha-beta).**
- **AI (alpha-beta) vs Random.**
- **Random vs Random.**

Are in the file (hexxagon.ipynb).

6.Main Loop

```python
# Create UI elements
dropdown1 = Dropdown(150, 100, 230, 40, player_options, select_player1)
dropdown2 = Dropdown(450, 100, 230, 40, player_options, select_player2)
start_button = Button(325, 300, 160, 50, "Start Game", start_game)

# Main loop
running = True
while running:
    screen.fill(WHITE)

    title = FONT.render("Hexxagon game", True, BLACK)
    screen.blit(title, (WIDTH // 2 - title.get_width() // 2, 30))

    dropdown1.draw(screen)
    dropdown2.draw(screen)

    if player1_selection and player2_selection:
        start_button.draw(screen)

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        dropdown1.check_event(event)
        dropdown2.check_event(event)
        start_button.check_event(event)

    pygame.display.flip()

pygame.quit()
```

TEAM MEMBERS: -

1.2301AI08-B. DHEERAJ.

2.2301AI09-M. DUNDI KULADEEPESHWAR.

3.2301AI29-P. TANUJ.

4.2301AI34-M. SATHVIK.

5.2301CS32-R. KARTHIKEYA.

6.2301CS64- YASHWANT REDDY.