**SRS Document for TheMineCheckers.py**

**Links:**

**Game Presentation - https://youtu.be/q9\_gCW0q0jM**

**PPT Presentation -** [**https://youtu.be/7zxTAhUK8TY**](https://youtu.be/7zxTAhUK8TY)

**GitHub: https://github.com/YuvalZin/AlgorithmProjectCS**

### **Table of Contents**

1. Introduction
   * Purpose
   * Scope
   * References
2. Overall Description
   * Product Perspective
   * Product Functions
   * Operating Environment
   * Design and Implementation Constraints
3. Specific Requirements
   * Functional Requirements
   * External Interface Requirements
   * System Features
4. Summary

### **Introduction**

#### **Purpose**

This SRS document provides a detailed description of the game "TheMineCheckers," a checkers-like game implemented using Python and the Pygame library. The purpose of this document is to outline the requirements, features, and specifications of the game.

#### **Scope**

The scope of this document includes the requirements and design of the game, which involves setting up a checkers-like board, placing pieces, adding mines, implementing game rules, and providing an AI opponent.

#### **References**

* [Pygame Documentation](https://www.pygame.org/docs/)
* [Python Documentation](https://docs.python.org/3/)

### **Overall Description**

#### **Product Perspective**

TheMineCheckers is a standalone application that provides a checkers-like gaming experience with the addition of mines on the board. The game allows for human and AI players to compete against each other to reach the end of the board.

#### **Product Functions**

* Initialize and display the game board with pieces and mines.
* Allow human players to move pieces.
* Implement AI to play against human players (using the minimax algorithm with alpha-beta pruning).
* Provide a reshuffle button to change the positions of the mines.
* Determine and display the game outcome (win/loss).

#### **Operating Environment**

* Operating System: Windows, macOS, Linux
* Python 3.x
* Pygame library - **Users must have Python and Pygame installed on their systems**.

#### **Design and Implementation Constraints**

* The game is designed using the Pygame library, which imposes constraints related to performance and graphical capabilities.
* The game logic must handle edge cases like no available moves and ensure that mines do not appear adjacent to each other in the same row.

### **Specific Requirements**

#### **Functional Requirements**

* **FR1**: The game shall initialize a 6x5 board with pieces for two players.
* **FR2**: The game shall allow players to move pieces according to game rules.
* **FR3**: The game shall place mines on the board in random locations, ensuring no two mines are adjacent in the same row.
* **FR4**: The game shall provide a reshuffle button to reposition the mines (2 times per game for a player).
* **FR5**: The game shall determine the winner based on reaching the opposite end of the board or eliminating all opponent pieces.
* **FR6**: The game shall implement AI to play against the human player.
* **FR7**: The AI implemented player shall use the Alpha Beta Pruning algorithm to choose its next move.
* **FR8**: The game shall display messages for win/loss conditions.

#### **External Interface Requirements**

* **GUI**: The game has a graphical user interface using Pygame for displaying the board, pieces, mines, and reshuffle button.

#### **System Features**

* **Board Setup**: Initialize and display a 6x5 board with player pieces.
* **Player Moves**: Allow players to select and move pieces on their turn.
* **Mine Placement**: Randomly place mines on the board, ensuring no two mines are adjacent in the same row.
* **Reshuffle**: Provide a button to reshuffle the positions of the mines.
* **AI Opponent**: Implement AI to play against the human player.
* **Random Start**: At the start of the game, there is a “coin toss” to determine who will start the game.
* **Win/Loss Conditions**: Determine and display win/loss based on game rules.

### **Summary**

**Problem Statement**: The goal was to create an engaging and strategic game that combines elements of checkers and minesweeper, providing a unique gaming experience with both human and AI opponents.

**Solution**: We developed "TheMineCheckers," a checkers-like game with an added challenge of mines on the board. The game includes a reshuffle feature, allowing players to change the positions of the mines strategically.

**Algorithms Used**:

* **Minimax Algorithm with Alpha-Beta Pruning**: This algorithm is used by the AI to evaluate the best possible moves. It helps in reducing the number of nodes evaluated by the minimax algorithm in its search tree, optimizing the decision-making process.

The development process involved implementing the game board, game rules, AI opponent, and user interface, ensuring smooth gameplay and a fair challenge for players.

### **Code:**

import pygame  
import sys  
import random  
from tkinter import messagebox, Tk  
  
# Initialize Pygame  
pygame.init()  
  
# Constants  
ROWS, COLS = 6, 5  
SQUARE\_SIZE = 100  
WIDTH, HEIGHT = COLS \* SQUARE\_SIZE, ROWS \* SQUARE\_SIZE + 50 # Extra space for reshuffle button  
  
# Colors  
WHITE = (255, 255, 255)  
BLACK = (0, 0, 0)  
RED = (255, 0, 0)  
BLUE = (0, 0, 255)  
GRAY = (128, 128, 128)  
YELLOW = (255, 255, 0)  
  
# Initialize the screen  
screen = pygame.display.set\_mode((WIDTH, HEIGHT))  
pygame.display.set\_caption('Checkers-like Game')  
  
# Fonts  
FONT = pygame.font.SysFont('Arial', 24)  
  
# Reshuffle button  
RESHUFFLE\_BTN = pygame.Rect(WIDTH // 2 - 50, HEIGHT - 50, 100, 40)  
RESHUFFLE\_BTN\_COLOR = (200, 200, 200)  
RESHUFFLE\_BTN\_TEXT = FONT.render('Reshuffle', True, BLACK)  
  
# Piece class representing a game piece  
class Piece:  
 PADDING = 15  
 OUTLINE = 2  
  
 def \_\_init\_\_(self, row, col, color):  
 self.row = row  
 self.col = col  
 self.color = color  
 self.x = 0  
 self.y = 0  
 self.calc\_pos()  
  
 def calc\_pos(self):  
 self.x = SQUARE\_SIZE \* self.col + SQUARE\_SIZE // 2  
 self.y = SQUARE\_SIZE \* self.row + SQUARE\_SIZE // 2  
  
 def draw(self, screen):  
 radius = SQUARE\_SIZE // 2 - self.PADDING  
 pygame.draw.circle(screen, GRAY, (self.x, self.y), radius + self.OUTLINE)  
 pygame.draw.circle(screen, self.color, (self.x, self.y), radius)  
  
 def move(self, row, col):  
 self.row = row  
 self.col = col  
 self.calc\_pos()  
  
# Create the initial board setup  
def create\_board():  
 board = []  
 for row in range(ROWS):  
 board.append([])  
 for col in range(COLS):  
 if row == 0:  
 board[row].append(Piece(row, col, BLUE))  
 elif row == ROWS - 1:  
 board[row].append(Piece(row, col, RED))  
 else:  
 board[row].append(0)  
 return board  
  
# Generate mines on the board  
def generate\_mines(board, num\_mines=4):  
 mines = set()  
 while len(mines) < num\_mines:  
 row = random.randint(1, ROWS - 2)  
 col = random.randint(0, COLS - 1)  
 if board[row][col] == 0:  
 # Ensure no two mines are adjacent in the same row  
 if all((row, col + offset) not in mines for offset in [-1, 1]):  
 mines.add((row, col))  
 return mines  
  
# Draw the game board  
def draw\_board(screen, board, mines, turn, red\_shuffles, blue\_shuffles):  
 screen.fill(WHITE)  
 for row in range(ROWS):  
 for col in range(COLS):  
 if (row, col) in mines:  
 pygame.draw.rect(screen, YELLOW, (col \* SQUARE\_SIZE, row \* SQUARE\_SIZE, SQUARE\_SIZE, SQUARE\_SIZE))  
 elif (row + col) % 2 == 0:  
 pygame.draw.rect(screen, BLACK, (col \* SQUARE\_SIZE, row \* SQUARE\_SIZE, SQUARE\_SIZE, SQUARE\_SIZE))  
 else:  
 pygame.draw.rect(screen, WHITE, (col \* SQUARE\_SIZE, row \* SQUARE\_SIZE, SQUARE\_SIZE, SQUARE\_SIZE))  
  
 for row in range(ROWS):  
 for col in range(COLS):  
 piece = board[row][col]  
 if piece != 0:  
 piece.draw(screen)  
  
 draw\_reshuffle\_button(screen, turn, red\_shuffles, blue\_shuffles)  
  
# Draw the reshuffle button  
def draw\_reshuffle\_button(screen, turn, red\_shuffles, blue\_shuffles):  
 pygame.draw.rect(screen, RESHUFFLE\_BTN\_COLOR, RESHUFFLE\_BTN)  
 screen.blit(RESHUFFLE\_BTN\_TEXT, (RESHUFFLE\_BTN.x + 5, RESHUFFLE\_BTN.y + 5))  
  
 # Display remaining reshuffles for each player  
 red\_text = FONT.render(f'Red Shuffles: {red\_shuffles}', True, RED)  
 blue\_text = FONT.render(f'Blue Shuffles: {blue\_shuffles}', True, BLUE)  
 screen.blit(red\_text, (10, HEIGHT - 30))  
 screen.blit(blue\_text, (WIDTH - blue\_text.get\_width() - 10, HEIGHT - 30))  
  
# Convert mouse position to board coordinates  
def get\_row\_col\_from\_mouse(pos):  
 x, y = pos  
 row = y // SQUARE\_SIZE  
 col = x // SQUARE\_SIZE  
 return row, col  
  
# Show a message box with the game result  
def show\_message(message):  
 root = Tk()  
 root.withdraw()  
 messagebox.showinfo("Game Over", message)  
 try:  
 root.destroy()  
 except:  
 pass  
  
# Check if there is a win condition on the board  
def check\_win\_condition(board):  
 for col in range(COLS):  
 if isinstance(board[0][col], Piece) and board[0][col].color == RED:  
 return "You won!"  
 if isinstance(board[ROWS-1][col], Piece) and board[ROWS-1][col].color == BLUE:  
 return "You lost!"  
 return None  
  
# Check if a player has lost all their pieces  
def check\_no\_pieces\_left(board):  
 red\_pieces = sum(isinstance(piece, Piece) and piece.color == RED for row in board for piece in row)  
 blue\_pieces = sum(isinstance(piece, Piece) and piece.color == BLUE for row in board for piece in row)  
 if red\_pieces == 0:  
 return "Red lost!"  
 if blue\_pieces == 0:  
 return "Blue lost!"  
 return None  
  
# Shuffle the mines on the board  
def shuffle\_mines(board, mines):  
 new\_mines = generate\_mines(board)  
 while len(new\_mines & mines) > 0:  
 new\_mines = generate\_mines(board)  
 return new\_mines  
  
# Evaluate the board state for the minimax algorithm  
def evaluate(board):  
 red\_pieces = sum(isinstance(piece, Piece) and piece.color == RED for row in board for piece in row)  
 blue\_pieces = sum(isinstance(piece, Piece) and piece.color == BLUE for row in board for piece in row)  
 return blue\_pieces - red\_pieces  
  
# Get valid moves for a piece  
def get\_valid\_moves(piece, board, mines):  
 if piece.color == RED:  
 directions = [(-1, 0), (-1, -1), (-1, 1), (0, -1), (0, 1)] # Forward, forward-left, forward-right, left, right for red  
 else:  
 directions = [(1, 0), (1, -1), (1, 1), (0, -1), (0, 1)] # Forward, forward-left, forward-right, left, right for blue  
 valid\_moves = []  
 for d in directions:  
 r, c = piece.row + d[0], piece.col + d[1]  
 if 0 <= r < ROWS and 0 <= c < COLS and (r, c) not in mines:  
 if board[r][c] == 0:  
 valid\_moves.append((r, c))  
 elif board[r][c].color != piece.color:  
 # Capture move  
 valid\_moves.append((r, c))  
 return valid\_moves  
  
# Minimax algorithm with alpha-beta pruning  
def minimax(board, depth, alpha, beta, maximizing\_player, mines):  
 win\_message = check\_win\_condition(board)  
 if depth == 0 or win\_message:  
 if win\_message == "You won!":  
 return float('-inf')  
 elif win\_message == "You lost!":  
 return float('inf')  
 return evaluate(board)  
  
 if maximizing\_player:  
 max\_eval = float('-inf')  
 for row in board:  
 for piece in row:  
 if isinstance(piece, Piece) and piece.color == BLUE:  
 for move in get\_valid\_moves(piece, board, mines):  
 new\_board = [list(r) for r in board]  
 new\_piece = Piece(move[0], move[1], piece.color)  
 new\_board[piece.row][piece.col], new\_board[move[0]][move[1]] = 0, new\_piece  
 eval = minimax(new\_board, depth - 1, alpha, beta, False, mines)  
 max\_eval = max(max\_eval, eval)  
 alpha = max(alpha, eval)  
 if beta <= alpha:  
 break  
 return max\_eval  
 else:  
 min\_eval = float('inf')  
 for row in board:  
 for piece in row:  
 if isinstance(piece, Piece) and piece.color == RED:  
 for move in get\_valid\_moves(piece, board, mines):  
 new\_board = [list(r) for r in board]  
 new\_piece = Piece(move[0], move[1], piece.color)  
 new\_board[piece.row][piece.col], new\_board[move[0]][move[1]] = 0, new\_piece  
 eval = minimax(new\_board, depth - 1, alpha, beta, True, mines)  
 min\_eval = min(min\_eval, eval)  
 beta = min(beta, eval)  
 if beta <= alpha:  
 break  
 return min\_eval  
  
# Determine the best move for the AI  
def best\_move(board, mines):  
 best\_moves = []  
 best\_eval = float('-inf')  
 for row in board:  
 for piece in row:  
 if isinstance(piece, Piece) and piece.color == BLUE:  
 for move in get\_valid\_moves(piece, board, mines):  
 new\_board = [list(r) for r in board]  
 new\_piece = Piece(move[0], move[1], piece.color)  
 new\_board[piece.row][piece.col], new\_board[move[0]][move[1]] = 0, new\_piece  
 eval = minimax(new\_board, 3, float('-inf'), float('inf'), False, mines)  
 if eval > best\_eval:  
 best\_eval = eval  
 best\_moves = [(piece, move)]  
 elif eval == best\_eval:  
 best\_moves.append((piece, move))  
 return random.choice(best\_moves) if best\_moves else None  
  
# Check for winning moves  
def find\_winning\_move(board, mines):  
 for row in board:  
 for piece in row:  
 if isinstance(piece, Piece) and piece.color == BLUE:  
 for move in get\_valid\_moves(piece, board, mines):  
 if move[0] == ROWS - 1: # Check if the move leads to a win  
 return piece, move  
 return None  
  
def blue\_shuffle\_condition(board, mines, eval\_value):  
 # Check if the red player is 2 rows from the goal row  
 for col in range(COLS):  
 if isinstance(board[2][col], Piece) and board[2][col].color == RED:  
 return True  
  
 # Check if the evaluate function returns a negative value  
 if eval\_value < 0:  
 return True  
  
 return False  
  
def main():  
 board = create\_board()  
 mines = generate\_mines(board)  
  
 # Coin flip to decide who starts  
 turn = random.choice([RED, BLUE])  
 start\_message = "Red" if turn == RED else "Blue"  
  
 # Display starting player message  
 screen.fill(WHITE)  
 start\_text = FONT.render(f"After tossing the coin, {start\_message} starts!", True, BLACK)  
 screen.blit(start\_text, (WIDTH // 2 - start\_text.get\_width() // 2, HEIGHT // 2 - start\_text.get\_height() // 2))  
 pygame.display.flip()  
 pygame.time.wait(2000) # Display the message for 2 seconds  
  
 selected\_piece = None  
 running = True  
 red\_shuffles = 2  
 blue\_shuffles = 2  
 blue\_reshuffled = False # To keep track of blue reshuffling once per turn  
  
 while running:  
 for event in pygame.event.get():  
 if event.type == pygame.QUIT:  
 running = False  
  
 if event.type == pygame.MOUSEBUTTONDOWN:  
 pos = pygame.mouse.get\_pos()  
 if RESHUFFLE\_BTN.collidepoint(pos):  
 if turn == RED and red\_shuffles > 0:  
 mines = shuffle\_mines(board, mines)  
 red\_shuffles -= 1  
 elif turn == BLUE and blue\_shuffles > 0 and not blue\_reshuffled:  
 eval\_value = evaluate(board)  
 if blue\_shuffle\_condition(board, mines, eval\_value):  
 mines = shuffle\_mines(board, mines)  
 blue\_shuffles -= 1  
 blue\_reshuffled = True # Mark blue reshuffled this turn  
 else:  
 row, col = get\_row\_col\_from\_mouse(pos)  
 piece = board[row][col]  
  
 if selected\_piece:  
 valid\_moves = get\_valid\_moves(selected\_piece, board, mines)  
 if (row, col) in valid\_moves:  
 board[selected\_piece.row][selected\_piece.col], board[row][col] = 0, selected\_piece  
 selected\_piece.move(row, col)  
 selected\_piece = None  
 turn = BLUE # Switch turn after player's move  
 blue\_reshuffled = False # Reset reshuffle tracker for blue  
 else:  
 selected\_piece = None # Deselect if clicked on invalid move  
  
 elif piece and piece.color == turn:  
 selected\_piece = piece  
  
 # After human player's move, check if it's AI's turn  
 if turn == BLUE:  
 eval\_value = evaluate(board)  
 if blue\_shuffle\_condition(board, mines, eval\_value) and blue\_shuffles > 0 and not blue\_reshuffled:  
 mines = shuffle\_mines(board, mines)  
 blue\_shuffles -= 1  
 blue\_reshuffled = True # Mark blue reshuffled this turn  
 else:  
 winning\_move = find\_winning\_move(board, mines)  
 if winning\_move:  
 piece, new\_pos = winning\_move  
 board[piece.row][piece.col], board[new\_pos[0]][new\_pos[1]] = 0, piece  
 piece.move(new\_pos[0], new\_pos[1])  
 if check\_win\_condition(board) == "You lost!":  
 show\_message("You lost!")  
 running = False  
 turn = RED # Switch turn after AI's move  
 else:  
 move = best\_move(board, mines)  
 if move:  
 piece, new\_pos = move  
 board[piece.row][piece.col], board[new\_pos[0]][new\_pos[1]] = 0, piece  
 piece.move(new\_pos[0], new\_pos[1])  
 turn = RED # Switch turn after AI's move  
 elif blue\_shuffles > 0 and not blue\_reshuffled: # If no valid move, reshuffle mines  
 mines = shuffle\_mines(board, mines)  
 blue\_shuffles -= 1  
 blue\_reshuffled = True # Mark blue reshuffled this turn  
  
 draw\_board(screen, board, mines, turn, red\_shuffles, blue\_shuffles) # Draw the board  
 pygame.display.flip()  
  
 win\_message = check\_win\_condition(board)  
 if win\_message and win\_message == "You won!":  
 show\_message(win\_message)  
 running = False  
  
 pieces\_left\_message = check\_no\_pieces\_left(board)  
 if pieces\_left\_message:  
 if pieces\_left\_message == "Red lost!":  
 show\_message("You lost!")  
 elif pieces\_left\_message == "Blue lost!":  
 show\_message("You won!")  
 running = False  
  
 pygame.quit()  
 sys.exit()  
  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()