AI PROJECT

ARIMAA game implemented in python.

ARIMAA is a two-player strategy board game that was designed to be playable with a standard chess set and difficult for computers while still being easy to learn and fun to play for humans.

#**Implementation**: We have implemented 2 versions:
(i) Human vs Minimax Algorithm
(ii) Heuristic vs Minimax Algorithm

#**Rules**: Arimaa is played on an 8×8 board with four trap squares. There are six kinds of pieces, ranging from elephant (strongest) to rabbit (weakest). Stronger pieces can push or pull weaker pieces, and stronger pieces freeze weaker pieces. Pieces can be captured by dislodging them onto a trap square when they have no orthogonally adjacent friendly pieces.

The two players, Gold and Silver, each control sixteen pieces. These are, in order from strongest to weakest: one elephant, one camel, two horses, two dogs, two cats, and eight rabbits. These may be represented by the king, queen, rooks, bishops, knights, and pawns respectively when one plays using a chess set.

#**Objective**: The main object of the game is to move a rabbit of one's own color onto the home rank of the opponent, which is known as a goal. Thus Gold wins by moving a gold rabbit to the eighth rank, and Silver wins by moving a silver rabbit to the first rank. However, because it is difficult to usher a rabbit to the goal line while the board is full of pieces, an intermediate objective is to capture opposing pieces by pushing them into the trap squares.

The game can also be won by capturing all of the opponent's rabbits (elimination) or by depriving the opponent of legal moves (immobilization). Compared to goal, these are uncommon.

#**Movement**: After the pieces are placed on the board, the players alternate turns, starting with Gold. A turn consists of making one to four steps. With each step a piece may move into an unoccupied square one space left, right, forward, or backward, except that rabbits may not step backward. The steps of a turn may be made by a single piece or distributed among several pieces in any order.

A turn must make a net change to the position. Thus one cannot, for example, take one step forward and one step back with the same piece, effectively passing the turn and evading zugzwang. Furthermore, one's turn may not create the same position with the same player to move as has been created twice before. This rule is similar to the situational super ko rule in the game of Go, which prevents endless loops, and is in contrast to chess where endless loops are considered draws. The prohibitions on passing and repetition make Arimaa a drawless game.

#**Pushing and pulling**: The second diagram, from the same game as the initial position above,[10] helps illustrate the remaining rules of movement.

A player may use two consecutive steps of a turn to dislodge an opposing piece with a stronger friendly piece which is adjacent in one of the four cardinal directions. For example, a player's dog may dislodge an opposing rabbit or cat, but not a dog, horse, camel, or elephant. The stronger piece may pull or push the adjacent weaker piece. When pulling, the stronger piece steps into an empty square, and the square it came from is occupied by the weaker piece. The silver elephant on d5 could step to d4 (or c5 or e5) and pull the gold horse from d6 to d5. When pushing, the weaker piece is moved to an adjacent empty square, and the square it came from is occupied by the stronger piece. The gold elephant on d3 could push the silver rabbit on d2 to e2 and then occupy d2. Note that the rabbit on d2 can't be pushed to d1, c2, or d3, because those squares are not empty.

Friendly pieces may not be dislodged. Also, a piece may not push and pull simultaneously. For example, the gold elephant on d3 could not simultaneously push the silver rabbit on d2 to e2 and pull the silver rabbit from c3 to d3. An elephant can never be dislodged, since there is nothing stronger.

#**Freezing**: A piece which is adjacent in any cardinal direction to a stronger opposing piece is frozen, unless it is also adjacent to a friendly piece. Frozen pieces may not be moved by the owner, but may be dislodged by the opponent. A frozen piece can freeze another still weaker piece. The silver rabbit on a7 is frozen, but the one on d2 is able to move because it is adjacent to a silver piece. Similarly the gold rabbit on b7 is frozen, but the gold cat on c1 is not. The dogs on a6 and b6 do not freeze each other because they are of equal strength. An elephant cannot be frozen, since there is nothing stronger, but an elephant can be blockaded.

#**Capturing**: A piece which enters a trap square is captured and removed from the game unless there is a friendly piece orthogonally adjacent. Silver could move to capture the gold horse on d6 by pushing it to c6 with the elephant on d5. A piece on a trap square is captured when all adjacent friendly pieces move away. Thus if the silver rabbit on c4 and the silver horse on c2 move away, voluntarily or by being dislodged, the silver rabbit on c3 will be captured.

Note that a piece may voluntarily step into a trap square, even if it is thereby captured. Also, the second step of a pulling maneuver is completed even if the piece doing the pulling is captured on the first step. For example, Silver could step the silver rabbit from f4 to g4 (so that it will no longer support pieces at f3), and then step the silver horse from f2 to f3, which captures the horse; the horse's move could still pull the gold rabbit from f1 to f2.

1. AI VS AI

Code→ import pygame

import os

import random

import math

from copy import deepcopy

```
import time

# Set up the game window
WINDOW_WIDTH = 600
WINDOW_HEIGHT = 640  # Extra height for displaying turn info
BOARD_SIZE = 8
CELL_SIZE = WINDOW_WIDTH // BOARD_SIZE  # Each square is 75 pixels

# Colors for the board
LIGHT_COLOR = (240, 217, 181)  # Beige
DARK_COLOR = (181, 136, 99)    # Brown
TRAP_COLOR = (255, 180, 60)    # Amber
HIGHLIGHT_COLOR = (200, 200, 100)  # Yellow

# Trap squares where pieces can be captured
TRAPS = [(2, 2), (2, 5), (5, 2), (5, 5)]

# Starting board (8x8 grid)
board = [
    ["SE", "SH", "ST", "SC", "SE", "SC", "SH", "SD"],
    ["SR", "SR", "SR", "SR", "SR", "SR", "SR", "SR"],
    [" ", " ", " ", " ", " ", " ", " ", " "],
    [" ", " ", " ", " ", " ", " ", " ", " "],
    [" ", " ", " ", " ", " ", " ", " ", " "],
    [" ", " ", " ", " ", " ", " ", " ", " "],
    ["GR", "GR", "GR", "GR", "GR", "GR", "GR", "GR"],
    ["GD", "GH", "GT", "GE", "GE", "GC", "GH", "GD"]
]

PIECE_IMAGES = {
    'GE': 'gold_elephant.png',
    'GC': 'gold_camel.png',
```

```python
    'GT': 'gold_cat.png',
    'GH': 'gold_horse.png',
    'GD': 'gold_dog.png',
    'GR': 'gold_rabbit.png',
    'SE': 'silver_elephant.png',
    'SC': 'silver_camel.png',
    'ST': 'silver_cat.png',
    'SH': 'silver_horse.png',
    'SD': 'silver_dog.png',
    'SR': 'silver_rabbit.png'
}

# Piece strengths (higher number = stronger piece)
# Piece strengths (higher number = stronger piece)
piece_strength = {
    "GE": 5, "GC": 4, "GH": 3, "GD": 2, "GT": 1, "GR": 0,
    "SE": 5, "SC": 4, "SH": 3, "SD": 2, "ST": 1, "SR": 0,
    " ": -1  # Empty space
}

# Game variables
whose_turn = "Gold"  # Gold goes first
move_count = 0  # How many moves made this turn
game_finished = False  # Is the game over?
move_history = []  # Store previous board states to detect loops
max_history_length = 10  # Keep the last 10 board states for loop detection

# Initialize pygame
pygame.init()
screen = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
pygame.display.set_caption("Arimaa: Minimax vs Heuristic")

def load_images():
```

```
    """Load images for each piece."""
    images = {}
    for piece, filename in PIECE_IMAGES.items():
        try:
            path = os.path.join(os.getcwd(), filename)
            images[piece] = pygame.image.load(path)
            images[piece] = pygame.transform.scale(images[piece], (CELL_SIZE - 10, CELL_SIZE - 10))
            print(f"Loaded image for {piece}: {filename}")
        except Exception as e:
            print(f"Failed to load image for {piece}: {e}")
            # Create a fallback colored square
            img = pygame.Surface((CELL_SIZE - 10, CELL_SIZE - 10), pygame.SRCALPHA)
            color = (218, 165, 32) if piece[0] == 'G' else (192, 192, 192)
            pygame.draw.rect(img, color, (0, 0, CELL_SIZE - 10, CELL_SIZE - 10))
            font = pygame.font.SysFont('Arial', 20, bold=True)
            text = font.render(piece, True, (0, 0, 0))
            text_rect = text.get_rect(center=(img.get_width()//2, img.get_height()//2))
            img.blit(text, text_rect)
            images[piece] = img
    return images

# Load piece images
piece_images = load_images()


def draw_board():
    """Draw the board and all pieces on it."""
    global board, piece_images

    # Draw the board squares
    for row in range(BOARD_SIZE):
        for col in range(BOARD_SIZE):
            # Set color: light or dark checkerboard pattern
            if (row + col) % 2 == 0:
```

```
            color = LIGHT_COLOR
        else:
            color = DARK_COLOR

        # Traps get a special highlight
        if (row, col) in TRAPS:
            color = TRAP_COLOR
            # Make trap squares slightly red-tinted
            # if (row + col) % 2 == 0:
            #     color = (min(255, LIGHT_COLOR[0] + 20), max(0, LIGHT_COLOR[1] - 30), max(0, LIGHT_COLOR[2] - 30))
            # else:
            #     color = (min(255, DARK_COLOR[0] + 20), max(0, DARK_COLOR[1] - 30), max(0, DARK_COLOR[2] - 30))

        # Draw the square
        pygame.draw.rect(screen, color, (col * CELL_SIZE, row * CELL_SIZE, CELL_SIZE, CELL_SIZE))

        # if (row, col) in TRAPS:
        #     pygame.draw.circle(screen, TRAP_COLOR,
        #                        (col * CELL_SIZE + CELL_SIZE // 2,
        #                         row * CELL_SIZE + CELL_SIZE // 2),
        #                        8, 2)  # Outlined circle

        # Draw the piece if present
        piece = board[row][col]
        if piece != "  " and piece in piece_images:
            img_rect = piece_images[piece].get_rect(
                center=(col * CELL_SIZE + CELL_SIZE // 2,
                        row * CELL_SIZE + CELL_SIZE // 2))
            screen.blit(piece_images[piece], img_rect)

    # Draw the grid lines
```

```
    # grid_color = (50, 50, 50)
    # for i in range(BOARD_SIZE + 1):
    #     # Horizontal lines
    #     pygame.draw.line(screen, grid_color,
    #                     (0, i * CELL_SIZE),
    #                     (WINDOW_WIDTH, i * CELL_SIZE), 1)
    #     # Vertical lines
    #     pygame.draw.line(screen, grid_color,
    #                     (i * CELL_SIZE, 0),
    #                     (i * CELL_SIZE, BOARD_SIZE * CELL_SIZE), 1)

    # Add row and column labels
    font = pygame.font.SysFont('Arial', 14)
    # for i in range(BOARD_SIZE):
    #     # Row labels (numbers)
    #     label = font.render(str(BOARD_SIZE - i), True, (150, 150, 150))
    #     screen.blit(label, (5, i * CELL_SIZE + 5))

    #     # Column labels (letters)
    #     label = font.render(chr(65 + i), True, (150, 150, 150))
    #     screen.blit(label, (i * CELL_SIZE + CELL_SIZE - 15, BOARD_SIZE * CELL_SIZE - 20))

    # # Display turn information and controls at bottom
    # font = pygame.font.SysFont('Arial', 18)

    # Turn info
    info_text = f"Turn: {whose_turn} ({'Minimax AI' if whose_turn == 'Gold' else 'Heuristic AI'}) - Moves:
{move_count}/4"
    info_surface = font.render(info_text, True, (255, 255, 255))
    screen.blit(info_surface, (10, BOARD_SIZE * CELL_SIZE + 10))

    # Controls
    controls_text = "Controls: SPACE to advance, R to restart, ESC to quit"
```

```
    controls_surface = font.render(controls_text, True, (200, 200, 200))
    screen.blit(controls_surface, (WINDOW_WIDTH - 400, BOARD_SIZE * CELL_SIZE + 10))

    # Show win message if game is over
    if game_finished:
        font = pygame.font.Font(None, 48)
        win_message = f"{whose_turn} Wins!"
        text = font.render(win_message, True, (255, 255, 255))
        text_pos = text.get_rect(center=(WINDOW_WIDTH // 2, WINDOW_HEIGHT // 2))
        pygame.draw.rect(screen, (0, 0, 0), text_pos.inflate(20, 20))  # Black background
        screen.blit(text, text_pos)

def is_frozen(board, row, col):
    """Check if a piece is frozen (surrounded by stronger enemy pieces)."""
    if board[row][col] == "  ":
        return False

    piece = board[row][col]
    player_prefix = piece[0]
    strength = piece_strength[piece]

    # Check if any adjacent position has a stronger enemy piece
    has_stronger_enemy = False
    for dr, dc in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
        adj_row, adj_col = row + dr, col + dc
        if 0 <= adj_row < BOARD_SIZE and 0 <= adj_col < BOARD_SIZE:
            adj_piece = board[adj_row][adj_col]
            if adj_piece != "  " and adj_piece[0] != player_prefix:
                if piece_strength[adj_piece] > strength:
                    has_stronger_enemy = True
                    break

    if not has_stronger_enemy:
```

```
        return False

    # Check if any adjacent position has a friendly piece
    for dr, dc in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
        adj_row, adj_col = row + dr, col + dc
        if 0 <= adj_row < BOARD_SIZE and 0 <= adj_col < BOARD_SIZE:
            adj_piece = board[adj_row][adj_col]
            if adj_piece != "  " and adj_piece[0] == player_prefix:
                return False  # Not frozen, has friendly support

    return True  # Frozen: has stronger enemy and no friendly support

def can_move(board, start_row, start_col, end_row, end_col):
    """Check if a piece can legally move from start to end."""
    # Check if coordinates are valid
    if not (0 <= start_row < BOARD_SIZE and 0 <= start_col < BOARD_SIZE):
        return False
    if not (0 <= end_row < BOARD_SIZE and 0 <= end_col < BOARD_SIZE):
        return False

    # Check if there's a piece at the start
    piece = board[start_row][start_col]
    if piece == "  ":
        return False

    # Check if the destination is empty
    if board[end_row][end_col] != "  ":
        return False

    # Check if move is orthogonal (no diagonals)
    if start_row != end_row and start_col != end_col:
        return False
```

```
    # Check if move is adjacent (no jumps)
    if abs(start_row - end_row) + abs(start_col - end_col) != 1:
        return False

    # Check if the piece is frozen
    if is_frozen(board, start_row, start_col):
        return False

    # Special rule for rabbits: cannot move backward
    if piece[1] == 'R':
        if piece[0] == 'G' and end_row > start_row:  # Gold rabbits can't move down
            return False
        if piece[0] == 'S' and end_row < start_row:  # Silver rabbits can't move up
            return False

    return True

def can_push_pull(board, piece_row, piece_col, target_row, target_col):
    """Check if a piece can push or pull the target piece."""
    # Check if coordinates are valid
    if not (0 <= piece_row < BOARD_SIZE and 0 <= piece_col < BOARD_SIZE):
        return False
    if not (0 <= target_row < BOARD_SIZE and 0 <= target_col < BOARD_SIZE):
        return False

    # Check if there's a piece at both positions
    piece = board[piece_row][piece_col]
    target = board[target_row][target_col]
    if piece == "  " or target == "  ":
        return False

    # Check if they're different colors
    if piece[0] == target[0]:
```

```
        return False

    # Check if they're adjacent
    if abs(piece_row - target_row) + abs(piece_col - target_col) != 1:
        return False

    # Check if the pushing/pulling piece is stronger
    if piece_strength[piece] <= piece_strength[target]:
        return False

    # Check if the piece is frozen
    if is_frozen(board, piece_row, piece_col):
        return False

    return True

def check_traps(board):
    """Check all trap squares and remove pieces without adjacent friendly pieces."""
    # Trap locations (row, col)
    traps = [(2, 2), (2, 5), (5, 2), (5, 5)]

    # Check each trap
    for trap_row, trap_col in traps:
        # If there's a piece on a trap
        if board[trap_row][trap_col] != "  ":
            piece = board[trap_row][trap_col]
            player_prefix = piece[0]  # 'G' or 'S'

            # Check if there's any adjacent friendly piece
            has_friendly_support = False
            for dr, dc in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
                adj_row, adj_col = trap_row + dr, trap_col + dc
```

```
            if 0 <= adj_row < BOARD_SIZE and 0 <= adj_col < BOARD_SIZE:
                adj_piece = board[adj_row][adj_col]
                if adj_piece != "  " and adj_piece[0] == player_prefix:
                    has_friendly_support = True
                    break

        # If no friendly support, the piece is captured
        if not has_friendly_support:
            #print(f"Piece {piece} captured at trap ({trap_row}, {trap_col})")
            board[trap_row][trap_col] = "  "  # Remove the piece

    return board

def check_winner(board):
    """Check if the game is won and set the winner."""
    global game_finished, whose_turn

    # Check for rabbit elimination
    gr_count = 0
    sr_count = 0
    for row in range(BOARD_SIZE):
        for col in range(BOARD_SIZE):
            if board[row][col] == 'GR':
                gr_count += 1
            if board[row][col] == 'SR':
                sr_count += 1

    # If all rabbits of one side are eliminated
    if gr_count == 0:
        print("All Gold rabbits eliminated - Silver wins!")
        whose_turn = 'Silver'
        game_finished = True
        return True
```

```
    elif sr_count == 0:
        print("All Silver rabbits eliminated - Gold wins!")
        whose_turn = 'Gold'
        game_finished = True
        return True

    # Check for rabbit reaching goal row
    for col in range(BOARD_SIZE):
        if board[0][col] == "GR":  # Gold rabbit at top row
            print("Gold rabbit reached the goal row - Gold wins!")
            print(col)
            whose_turn = "Gold"
            game_finished = True
            return True
        if board[7][col] == "SR":  # Silver rabbit at bottom row
            print("Silver rabbit reached the goal row - Silver wins!")
            print(col)
            whose_turn = "Silver"
            game_finished = True
            return True

    return False

def heuristic(board, add_noise=False):
    """Evaluate the board position from Gold's perspective."""
    h = 0

    # Piece value weights
    piece_values = {
        'SE': 100, 'SC': 50, 'SH': 30, 'SD': 20, 'ST': 15, 'SR': 10,
        'GE': -100, 'GC': -50, 'GH': -30, 'GD': -20, 'GT': 15, 'GR': -10,
        ' ': 0
    }
```

```
# Count material
for row in range(BOARD_SIZE):
    for col in range(BOARD_SIZE):
        piece = board[row][col]
        h += piece_values.get(piece, 0)

# Rabbit advancement - Silver rabbits want to go down, Gold rabbits want to go up
for row in range(BOARD_SIZE):
    for col in range(BOARD_SIZE):
        piece = board[row][col]
        if piece == 'SR':
            # Exponential reward for advancement
            h += (row + 1) ** 2
            # Extra bonus for being close to the goal row
            if row == 7:
                h = float('inf')  # Win condition
            elif row >= 6:  # One step away from winning
                h += 200
            elif row >= 5:  # Two steps away
                h += 100

        elif piece == 'GR':
            # Penalize Gold rabbit advancement (since this is from Silver's perspective)
            h -= (8 - row) ** 2
            if row == 0:
                h = -float('inf')  # Loss condition

# Control of center - pieces in the center have more influence
center_value = [
    [1, 1, 2, 2, 2, 2, 1, 1],
    [1, 2, 3, 3, 3, 3, 2, 1],
    [2, 3, 4, 4, 4, 4, 3, 2],
```

```
    [2, 3, 4, 5, 5, 4, 3, 2],
    [2, 3, 4, 5, 5, 4, 3, 2],
    [2, 3, 4, 4, 4, 4, 3, 2],
    [1, 2, 3, 3, 3, 3, 2, 1],
    [1, 1, 2, 2, 2, 2, 1, 1]
]

for row in range(BOARD_SIZE):
    for col in range(BOARD_SIZE):
        piece = board[row][col]
        if piece.startswith('S'):
            h += center_value[row][col] * 2
        elif piece.startswith('G'):
            h -= center_value[row][col] * 2

# Trap control and piece safety
for trap_row, trap_col in TRAPS:
    silver_adjacent = 0
    gold_adjacent = 0

    for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
        r, c = trap_row + dr, trap_col + dc
        if 0 <= r < BOARD_SIZE and 0 <= c < BOARD_SIZE:
            piece = board[r][c]
            if piece.startswith('S'):
                silver_adjacent += 1
            elif piece.startswith('G'):
                gold_adjacent += 1

    # Reward for controlling trap
    if silver_adjacent > gold_adjacent:
        h += 15 * (silver_adjacent - gold_adjacent)
    elif gold_adjacent > silver_adjacent:
```

```
        h -= 15 * (gold_adjacent - silver_adjacent)

    # Check pieces in traps
    piece_in_trap = board[trap_row][trap_col]
    if piece_in_trap != " ":
        if piece_in_trap.startswith('S') and silver_adjacent == 0:
            h -= 50  # Severe penalty for unsupported piece in trap
        elif piece_in_trap.startswith('G') and gold_adjacent == 0:
            h += 50  # Reward for enemy piece about to be captured

# File control - reward controlling files (columns)
for col in range(BOARD_SIZE):
    silver_count = 0
    gold_count = 0
    for row in range(BOARD_SIZE):
        piece = board[row][col]
        if piece.startswith('S'):
            silver_count += 1
        elif piece.startswith('G'):
            gold_count += 1

    # Reward for controlling files
    if silver_count > gold_count:
        h += 10 * (silver_count - gold_count)
    elif gold_count > silver_count:
        h -= 10 * (gold_count - silver_count)

# Piece mobility and safety
for row in range(BOARD_SIZE):
    for col in range(BOARD_SIZE):
        piece = board[row][col]
        if piece != " ":
            # Count possible moves for this piece
```

```
            moves = 0
            for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                r, c = row + dr, col + dc
                if 0 <= r < BOARD_SIZE and 0 <= c < BOARD_SIZE:
                    if board[r][c] == " " and not is_frozen(board, row, col):
                        moves += 1

            # Reward mobility
            if piece.startswith('S'):
                h += moves * 2
            else:
                h -= moves * 2

    # Add a small amount of noise to prevent repetitive patterns
    if add_noise:
        h += random.uniform(-20, 20)

    return h

def debug_moves(board, player):
    """Debug function to print available moves."""
    moves = generate_moves(board, player)
    print(f"\nMoves available for {player}: {len(moves)}")

    if len(moves) > 0:
        print("First 10 moves:")
        count = 0
        for move in moves:
            if move[0] != "pass":
                print(f" - {move}")
                count += 1
                if count >= 10:
                    break
```

```
    # Check frozen pieces
    print("\nFrozen pieces:")
    frozen_count = 0
    for row in range(BOARD_SIZE):
        for col in range(BOARD_SIZE):
            piece = board[row][col]
            if piece != " " and piece[0] == player[0]:
                if is_frozen(board, row, col):
                    print(f" - {piece} at ({row}, {col}) is FROZEN")
                    frozen_count += 1

    if frozen_count == 0:
        print(" - None")

    return moves

def generate_moves(board, current_turn, move_count=0):
    """Generate all possible moves for the current player."""
    moves = []

    # Only generate moves if we haven't used all 4 moves
    if move_count < 4:
        # Generate regular moves
        for row in range(BOARD_SIZE):
            for col in range(BOARD_SIZE):
                piece = board[row][col]
                if piece != " " and piece[0] == current_turn[0]:
                    # Check normal moves
                    for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                        new_row = row + dr
                        new_col = col + dc
                        if can_move(board, row, col, new_row, new_col):
```

```
            moves.append((row, col, new_row, new_col, "move"))

    # Push/pull moves - require 2 moves so only if < 3 moves used
    if move_count < 3:
        # Check for adjacent enemy pieces that can be pushed/pulled
        for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            adj_row = row + dr
            adj_col = col + dc
            if can_push_pull(board, row, col, adj_row, adj_col):
                # Try push directions
                for pdr, pdc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                    push_row = adj_row + pdr
                    push_col = adj_col + pdc
                    if 0 <= push_row < BOARD_SIZE and 0 <= push_col < BOARD_SIZE:
                        if board[push_row][push_col] == " ":
                            # Check if push would create a trap capture
                            temp_board = [row[:] for row in board]
                            temp_board[push_row][push_col] = temp_board[adj_row][adj_col]
                            temp_board[adj_row][adj_col] = temp_board[row][col]
                            temp_board[row][col] = " "
                            check_traps(temp_board)

                            # If push would capture a piece, prioritize it
                            if temp_board[push_row][push_col] == " ":
                                moves.append((row, col, adj_row, adj_col, "push", pdr, pdc))
                            else:
                                # Add with higher priority
                                moves.append((row, col, adj_row, adj_col, "push", pdr, pdc))

                # Try pull directions
                for pdr, pdc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                    # Skip direction toward the enemy piece
                    if pdr == dr and pdc == dc:
```

```
                    continue

                pull_row = row + pdr
                pull_col = col + pdc
                if 0 <= pull_row < BOARD_SIZE and 0 <= pull_col < BOARD_SIZE:
                    if board[pull_row][pull_col] == " ":
                        # Check if pull would create a trap capture
                        temp_board = [row[:] for row in board]
                        temp_board[pull_row][pull_col] = temp_board[row][col]
                        temp_board[row][col] = temp_board[adj_row][adj_col]
                        temp_board[adj_row][adj_col] = " "
                        check_traps(temp_board)

                        # If pull would capture a piece, prioritize it
                        if temp_board[row][col] == " ":
                            moves.append((row, col, adj_row, adj_col, "pull", pdr, pdc))
                        else:
                            # Add with higher priority
                            moves.append((row, col, adj_row, adj_col, "pull", pdr, pdc))

        # Add "pass" move if at least one move was made
        if move_count >= 1:
            moves.append(("pass", None, None, None, None))

    # Debug information
    if len(moves) == 0:
        print(f"WARNING: No valid moves generated for {current_turn}")

    return moves

def make_move(board, move):
    """Apply a move to the board and return the new board."""
    if move[0] == "pass":
```

```
    return [row[:] for row in board]  # Return a copy of the board

# Create a copy of the board to modify
new_board = [row[:] for row in board]

if move[4] == "move":
    start_row, start_col, end_row, end_col, _ = move
    new_board[end_row][end_col] = new_board[start_row][start_col]
    new_board[start_row][start_col] = " "

elif move[4] == "push":
    start_row, start_col, end_row, end_col, _, dir_row, dir_col = move
    push_row, push_col = end_row + dir_row, end_col + dir_col

    # Move the opponent's piece first
    new_board[push_row][push_col] = new_board[end_row][end_col]
    # Then move our piece to opponent's previous spot
    new_board[end_row][end_col] = new_board[start_row][start_col]
    # Empty our original position
    new_board[start_row][start_col] = " "

elif move[4] == "pull":
    start_row, start_col, end_row, end_col, _, dir_row, dir_col = move
    pull_row, pull_col = start_row + dir_row, start_col + dir_col

    # Move our piece first
    new_board[pull_row][pull_col] = new_board[start_row][start_col]
    # Then move opponent's piece to our original spot
    new_board[start_row][start_col] = new_board[end_row][end_col]
    # Empty opponent's original position
    new_board[end_row][end_col] = " "

# Check traps after any move
```

```
    check_traps(new_board)

    return new_board

def board_to_string(board):
    """Convert a board to a string representation for loop detection."""
    return "".join("".join(row) for row in board)

def is_loop_detected(board_history):
    """Check if the current board state has appeared multiple times."""
    if len(board_history) < 6:
        return False

    # Get the last board state
    last_state = board_history[-1]

    # Count occurrences of this board state in history
    occurrences = sum(1 for state in board_history if state == last_state)

    # If this board state has appeared 3+ times, it's a loop
    return occurrences >= 3

def add_to_history(board):
    """Add the current board state to history."""
    global move_history
    board_str = board_to_string(board)
    move_history.append(board_str)
    # Keep only the last max_history_length states
    if len(move_history) > max_history_length:
        move_history.pop(0)

def handle_ai_turn():
    """Handle the AI's turn (up to 4 moves)."""
```

```
global whose_turn, move_count, game_finished, board, move_history

# Print debug info
print(f"\n{whose_turn}'s turn (move {move_count}/4):")
debug_moves(board, whose_turn)

# Check if game is already finished
if check_winner(board):
    return

# Track board states to detect loops
board_str = board_to_string(board)
if board_str not in move_history:
    move_history.append(board_str)
    if len(move_history) > max_history_length:
        move_history.pop(0)

# Check for loops in game
if is_loop_detected(move_history):
    print("Loop detected! Introducing randomness in move selection.")
    # Reset history to break the loop
    move_history = []

# Get the AI's move (Gold = Minimax, Silver = Heuristic)
if whose_turn == "Gold":
    best_move = get_best_move(board, "Gold")
else:
    best_move = find_best_move_heuristic(board)

# If no valid move or pass, end turn
if best_move is None or best_move[0] == "pass":
    print(f"{whose_turn} passes their turn")
    move_count = 4  # Force end of turn
```

```
    else:
        # Apply the move
        print(f"{whose_turn} makes move: {best_move}")
        board = make_move(board, best_move)
        move_count += 1

        # Check if game is over after the move
        if check_winner(board):
            return

    # Check if turn is over (all 4 moves used)
    if move_count >= 4:
        move_count = 0
        whose_turn = "Silver" if whose_turn == "Gold" else "Gold"

        # Check if game is over
        check_winner(board)

def minimax(board, depth, alpha, beta, maximizing_player, current_turn):
    """Minimax algorithm with alpha-beta pruning."""
    # Base case: depth limit reached or terminal node
    if depth == 0 or check_winner(board):
        return heuristic(board, add_noise=(depth == 0)), None

    # Generate all possible moves
    moves = generate_moves(board, current_turn)

    # No valid moves
    if not moves:
        return heuristic(board), None

    # Shuffle moves for more variety when scores are equal
```

```
    random.shuffle(moves)                                   ##################
Anchor

    if maximizing_player == current_turn:  # (maximizing)                  ##############
Anchor
        max_eval = float('-inf')
        best_move = None

        for move in moves:
            # Make the move
            new_board = make_move(board, move)

            # Recursively evaluate
            eval_score, _ = minimax(new_board, depth - 1, alpha, beta, False, "Gold")

            # Update best move if this is better
            if eval_score > max_eval:
                max_eval = eval_score
                best_move = move

            # Alpha-beta pruning
            alpha = max(alpha, eval_score)
            if beta <= alpha:
                break

        return max_eval, best_move

    else:  # Gold's turn (minimizing)
        min_eval = float('inf')
        best_move = None

        for move in moves:
            # Make the move
```

```python
            new_board = make_move(board, move)

            # Recursively evaluate
            eval_score, _ = minimax(new_board, depth - 1, alpha, beta, True, "Silver")

            # Update best move if this is better
            if eval_score < min_eval:
                min_eval = eval_score
                best_move = move

            # Alpha-beta pruning
            beta = min(beta, eval_score)
            if beta <= alpha:
                break

        return min_eval, best_move

def get_best_move(board, current_turn):
    """Find the best move using minimax with alpha-beta pruning."""
    # Get all available moves
    moves = generate_moves(board, current_turn)

    # If no valid moves or only pass, return None or pass
    if len(moves) <= 1:
        return None if len(moves) == 0 else moves[0]

    # Filter out pass move unless it's the only option
    non_pass_moves = [m for m in moves if m[0] != "pass"]
    if len(non_pass_moves) == 0:
        return moves[0]  # Only pass move available

    # If loop is detected, choose a random move
    if is_loop_detected([board_to_string(board)]):
```

```
        print("Loop detected in minimax - choosing random move")
        return random.choice(non_pass_moves)

    # Adjust search depth based on game complexity
    piece_count = sum(1 for row in board for piece in row if piece != " ")
    depth = 2

    # Deeper search for endgame positions with fewer pieces
    if piece_count < 10:
        depth = 3

    start_time = time.time()
    score, best_move = minimax(board, depth, float('-inf'), float('inf'), current_turn == "Silver",
current_turn)
    end_time = time.time()

    print(f"Minimax search (depth {depth}) took {end_time - start_time:.2f} seconds, score: {score}")

    # If minimax fails to find a move or returns pass, pick a random non-pass move
    if best_move is None or best_move[0] == "pass":
        if len(non_pass_moves) > 0:
            print("Minimax defaulting to random non-pass move")
            best_move = random.choice(non_pass_moves)

    return best_move

def find_best_move_heuristic(board):
    """Find the best move using a simple heuristic evaluation."""
    # Get all available moves
    moves = generate_moves(board, "Silver")

    # If no valid moves, return None
    if len(moves) <= 1:  # Only pass or no moves
```

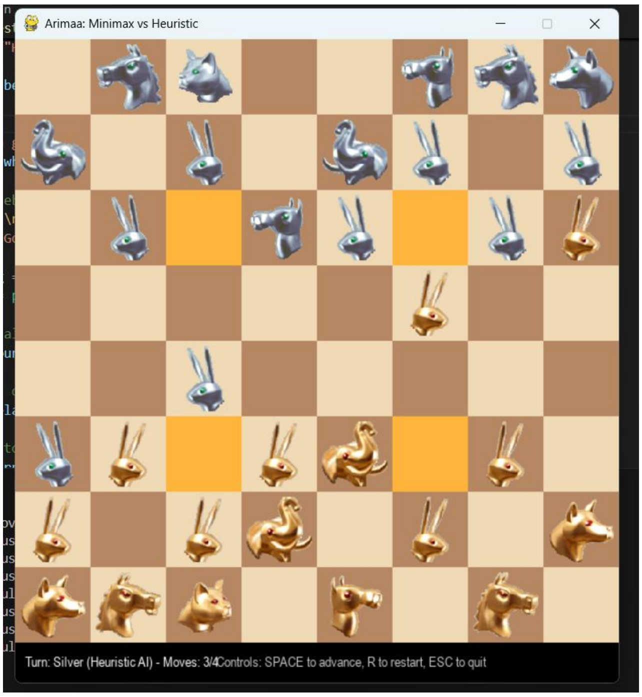
```
    return None if len(moves) == 0 else moves[0]

# Filter out pass move unless it's the only option
non_pass_moves = [m for m in moves if m[0] != "pass"]
if len(non_pass_moves) == 0:
    return moves[0]  # Only pass move available

# If loop is detected, choose a random move
if is_loop_detected([board_to_string(board)]):
    print("Loop detected in heuristic - choosing random move")
    return random.choice(non_pass_moves)

# Group moves by score for random selection among equal scores
move_scores = {}

# Evaluate each move
for move in non_pass_moves:
    # Make the move
    new_board = make_move(board, move)

    # Evaluate position with some noise for variety
    score = heuristic(new_board, add_noise=True)

    # Store by score
    if score not in move_scores:
        move_scores[score] = []
    move_scores[score].append(move)

# Find the best score (highest for Silver)
best_score = max(move_scores.keys())

# Choose randomly from the moves with the best score
best_move = random.choice(move_scores[best_score])
```

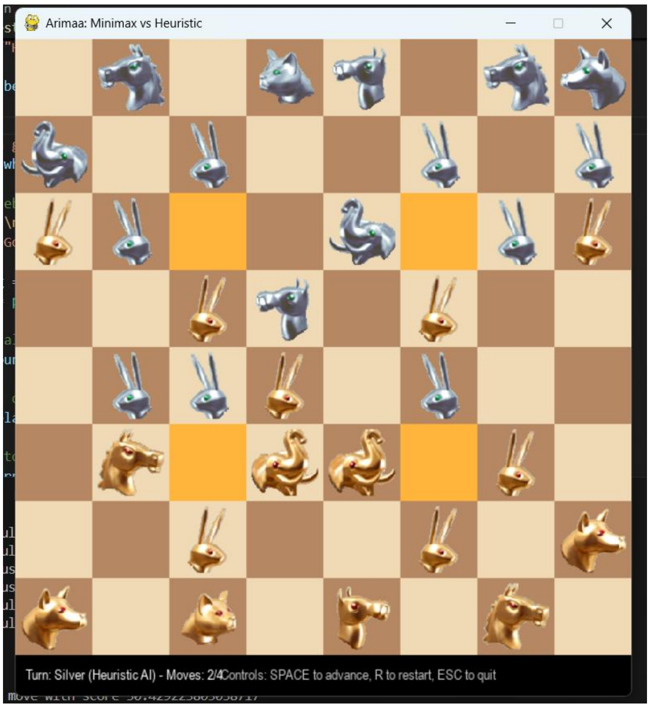
```
    print(f"Heuristic selected move with score {best_score}")

    return best_move

def main():
    """Main game loop."""
    global whose_turn, move_count, game_finished, board, move_history

    # For debugging: print initial state
    print("\n=== Starting Arimaa AI vs AI game ===")
    print("Gold = Minimax AI, Silver = Heuristic AI")

    running = True
    clock = pygame.time.Clock()

    # Initialize turn counter for display
    turn_counter = 1

    # Add a delay between turns for better visualization
    turn_delay = 1000  # 1 second delay between turns

    # Flag to control when to process the next turn
    next_turn_ready = True

    while running:
        # Process events
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_ESCAPE:
                    running = False
                # Spacebar to advance turns quickly
```

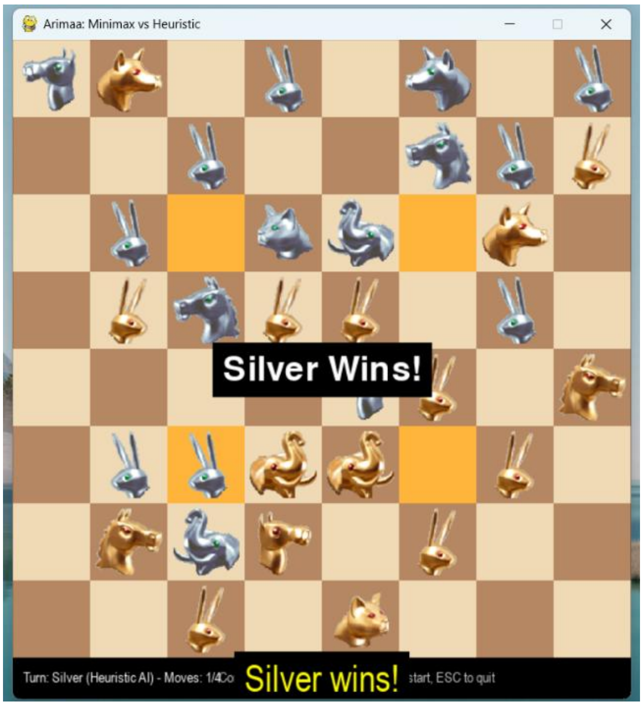
```
            elif event.key == pygame.K_SPACE and not game_finished:
                next_turn_ready = True
            # R key to restart the game
            elif event.key == pygame.K_r:
                # Reset game state
                board = [
                    ["SE", "SH", "ST", "SC", "SE", "SC", "SH", "SD"],
                    ["SR", "SR", "SR", "SR", "SR", "SR", "SR", "SR"],
                    [" ", " ", " ", " ", " ", " ", " ", " "],
                    [" ", " ", " ", " ", " ", " ", " ", " "],
                    [" ", " ", " ", " ", " ", " ", " ", " "],
                    [" ", " ", " ", " ", " ", " ", " ", " "],
                    ["GR", "GR", "GR", "GR", "GR", "GR", "GR", "GR"],
                    ["GD", "GH", "GT", "GE", "GE", "GC", "GH", "GD"]
                ]
                whose_turn = "Gold"
                move_count = 0
                game_finished = False
                turn_counter = 1
                move_history = []
                next_turn_ready = True
                print("\n=== Game restarted ===")

    # Clear screen and draw board
    screen.fill((0, 0, 0))
    draw_board()

    # AI gameplay
    if not game_finished and next_turn_ready:
        next_turn_ready = False  # Mark turn as in progress

        # Track whose turn it was
        current_turn = whose_turn
```

```
    # Let the AI play its turn
    handle_ai_turn()

    # If the turn changed, increment counter
    if whose_turn != current_turn:
        turn_counter += 1
        # Add delay between turns for better visualization
        pygame.time.delay(turn_delay)

    # Mark next turn as ready
    next_turn_ready = True

    # If we've been playing for a long time with no winner (200+ turns), declare a draw
    if turn_counter > 200:
        print("Game ended in a draw after 200 turns")
        game_finished = True

# Game over display
if game_finished:
    # Draw a centered game over message
    font = pygame.font.SysFont('Arial', 36)
    if turn_counter > 200:
        message = "Game ended in a draw!"
    else:
        message = f"{whose_turn} wins!"

    text = font.render(message, True, (255, 255, 0))
    text_rect = text.get_rect(center=(WINDOW_WIDTH // 2, WINDOW_HEIGHT - 20))

    # Draw a background for the text
    pygame.draw.rect(screen, (0, 0, 0), text_rect.inflate(20, 10))
    screen.blit(text, text_rect)
```

```
        # Update display
        pygame.display.flip()

        # Cap the frame rate
        clock.tick(30)

    # Clean up
    pygame.quit()
    print("\n=== Game ended ===")
    if game_finished:
        if turn_counter > 200:
            print("Game ended in a draw")
        else:
            print(f"Winner: {whose_turn}")
    else:
        print("Game closed without finishing")

# Start the game
if __name__ == "__main__":
    main()
```


Arimaa: Minimax vs Heuristic
Turn: Silver (Heuristic AI) - Moves: 3/4Controls: SPACE to advance, R to restart, ESC to quit


Arimaa: Minimax vs Heuristic
Turn: Silver (Heuristic AI) - Moves: 2/4Controls: SPACE to advance, R to restart, ESC to quit


Arimaa: Minimax vs Heuristic
Silver Wins!
Turn: Silver (Heuristic AI) - Moves: 1/4
Silver wins!
start, ESC to quit


Arimaa: Minimax vs Heuristic
Gold Wins!
Turn: Gold (Minimax AI) - Moves: 4/4
Gold wins!
start, ESC to quit

2. HUMAN VS AI (MINIMAX)

Code→ import pygame

```
import os
import random
import math
from copy import deepcopy
import time

# Set up the game window
WINDOW_WIDTH = 600
WINDOW_HEIGHT = 600
BOARD_SIZE = 8
CELL_SIZE = WINDOW_WIDTH // BOARD_SIZE  # Each square is 75 pixels

# Colors for the board
LIGHT_COLOR = (240, 217, 181)  # Beige
DARK_COLOR = (181, 136, 99)    # Brown
TRAP_COLOR = (255, 180, 60)    # Amber
HIGHLIGHT_COLOR = (200, 200, 100)  # Yellow

# Trap squares where pieces can be captured
TRAPS = [(2, 2), (2, 5), (5, 2), (5, 5)]

# Starting board (8x8 grid)
board = [
    ["SE", "SH", "SD", "SD", "SCT", "SCT", "SH", "SC"],
    ["SR", "SR", "SR", "SR", "SR", "SR", "SR", "SR"],
    [" ", " ", " ", " ", " ", " ", " ", " "],
    [" ", " ", " ", " ", " ", " ", " ", " "],
    [" ", " ", " ", " ", " ", " ", " ", " "],
    [" ", " ", " ", " ", " ", " ", " ", " "],
    ["GR", "GR", "GR", "GR", "GR", "GR", "GR", "GR"],
```

```
    ["GE", "GH", "GD", "GD", "GCT", "GCT", "GH", "GC"]
]

PIECE_IMAGES = {
    'GE': 'gold_elephant.png',
    'GC': 'gold_camel.png',
    'GH': 'gold_horse.png',
    'GD': 'gold_dog.png',
    'GCT': 'gold_cat.png',
    'GR': 'gold_rabbit.png',
    'SE': 'silver_elephant.png',
    'SC': 'silver_camel.png',
    'SH': 'silver_horse.png',
    'SD': 'silver_dog.png',
    'SCT': 'silver_cat.png',
    'SR': 'silver_rabbit.png'
}

# Piece strengths (higher number = stronger piece)
piece_strength = {
    "GE": 5, "GC": 4, "GH": 3, "GD": 2, "GCT": 1, "GR": 0,
    "SE": 5, "SC": 4, "SH": 3, "SD": 2, "SCT": 1, "SR": 0,
    " ": -1  # Empty space
}

def load_images():
    images = {}
    for piece, filename in PIECE_IMAGES.items():
        path = os.path.join(os.getcwd(), filename)
        images[piece] = pygame.image.load(path)
        images[piece] = pygame.transform.scale(images[piece], (CELL_SIZE - 10, CELL_SIZE - 10))
    return images
```

```
# Start the game
pygame.init()
piece_images = load_images()
screen = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
pygame.display.set_caption("Arimaa: Human vs Minimax AI")

# Game variables
selected = None  # What's clicked: None, (row, col), or ((r1, c1), (r2, c2))
whose_turn = "Gold"  # Whose turn it is
move_count = 0  # How many moves made this turn
game_finished = False  # Is the game over?

def draw_board():
    # Loop through each square
    for row in range(BOARD_SIZE):
        for col in range(BOARD_SIZE):
            # Set color: light or dark checkerboard pattern
            if (row + col) % 2 == 0:
                color = LIGHT_COLOR
            else:
                color = DARK_COLOR

            # Traps get a special color
            if (row, col) in TRAPS:
                color = TRAP_COLOR

            # Highlight selected squares if game isn't over
            if selected and not game_finished:
                # Single piece selected
                if type(selected) == tuple and len(selected) == 2 and type(selected[0]) == int:
                    if (row, col) == selected:
                        color = HIGHLIGHT_COLOR
                # Two pieces selected for push/pull
```

```
        elif type(selected) == tuple and len(selected) == 2 and type(selected[0]) == tuple:
            if (row, col) == selected[0] or (row, col) == selected[1]:
                color = HIGHLIGHT_COLOR

        # Draw the square
        pygame.draw.rect(screen, color, (col * CELL_SIZE, row * CELL_SIZE, CELL_SIZE,
CELL_SIZE))

        # Draw the piece
        piece = board[row][col]
        if piece in piece_images:
            img_rect = piece_images[piece].get_rect(center=(col * CELL_SIZE + CELL_SIZE / 2, row *
CELL_SIZE + CELL_SIZE / 2))
            screen.blit(piece_images[piece], img_rect)

    # Show win message if game is over
    if game_finished:
        font = pygame.font.Font(None, 48)
        win_message = f"{whose_turn} Wins!"
        text = font.render(win_message, True, (255, 255, 255))
        text_pos = text.get_rect(center=(WINDOW_WIDTH // 2, WINDOW_HEIGHT // 2))
        pygame.draw.rect(screen, (0, 0, 0), text_pos.inflate(20, 20))  # Black background
        screen.blit(text, text_pos)

def is_frozen(row, col, board):
    piece = board[row][col]
    if piece == " ":
        return False

    frozen = False
    has_friend = False

    # Check all 4 directions: up, down, left, right
```

```
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for dir_row, dir_col in directions:
        new_row = row + dir_row
        new_col = col + dir_col
        # Make sure we're still on the board
        if 0 <= new_row < BOARD_SIZE and 0 <= new_col < BOARD_SIZE:
            nearby_piece = board[new_row][new_col]
            if nearby_piece != " ":
                # Friend nearby (same team)?
                if nearby_piece[0] == piece[0]:
                    has_friend = True
                # Stronger enemy nearby?
                elif piece_strength[piece] < piece_strength[nearby_piece]:
                    frozen = True

    # Frozen only if there's a stronger enemy and no friends
    return frozen and not has_friend

def can_move(start_row, start_col, end_row, end_col, board = board):
    # Must be on the board and to an empty space
    if not (0 <= end_row < BOARD_SIZE and 0 <= end_col < BOARD_SIZE):
        return False
    if board[end_row][end_col] != " " or is_frozen(start_row, start_col, board):
        return False

    piece = board[start_row][start_col]
    row_change = start_row - end_row
    col_change = abs(start_col - end_col)

    # Must move exactly one step
    if abs(row_change) + col_change != 1:
        return False
```

```
    # Rabbits have special rules
    if piece == "GR":  # Gold rabbit: backward or sideways
        return row_change == 1 or col_change == 1
    if piece == "SR":  # Silver rabbit: forward or sideways
        return row_change == -1 or col_change == 1
    return True

def can_push_or_pull(start_row, start_col, end_row, end_col, board = board):
    if 0 <= start_row < 8 and 0 <= start_col < 8 and 0 <= end_row < 8 and 0 <= end_col < 8:
        piece = board[start_row][start_col]
        target = board[end_row][end_col]
    else:
        return False

    # Must be next to each other
    if abs(start_row - end_row) + abs(start_col - end_col) != 1:
        return False

    # Gold pushing/pulling Silver, or Silver pushing/pulling Gold
    if piece[0] == "G" and target[0] == "S" and piece_strength[piece] > piece_strength[target]:
        return True
    if piece[0] == "S" and target[0] == "G" and piece_strength[piece] > piece_strength[target]:
        return True
    return False

def push(start_row, start_col, end_row, end_col, dir_row, dir_col, board = board):
    global move_count
    new_row = end_row + dir_row
    new_col = end_col + dir_col

    # Check if the new spot is on the board and empty
    if not (0 <= new_row < BOARD_SIZE and 0 <= new_col < BOARD_SIZE):
        return False  # Can't push out of bounds
```

```python
    if is_frozen(start_row, start_col, board) or board[new_row][new_col] != " ":
        return False  # Destination must be empty

    # Ensure the pusher is stronger than the pushed piece
    if piece_strength[board[start_row][start_col]] <= piece_strength[board[end_row][end_col]]:
        return False

    # Move the pushed piece
    board[new_row][new_col] = board[end_row][end_col]
    # Move the pusher to the pushed piece's old spot
    board[end_row][end_col] = board[start_row][start_col]
    # Empty the pusher's original spot
    board[start_row][start_col] = " "

    move_count += 2
    return True

def pull(start_row, start_col, end_row, end_col, dir_row, dir_col, board = board):
    global move_count
    new_row = start_row + dir_row  # The puller moves in the opposite direction of the pull
    new_col = start_col + dir_col

    # Check if the new spot is on the board and empty
    if not (0 <= new_row < BOARD_SIZE and 0 <= new_col < BOARD_SIZE):
        return False
    if is_frozen(start_row, start_col, board) or board[new_row][new_col] != " ":
        return False

    if piece_strength[board[start_row][start_col]] <= piece_strength[board[end_row][end_col]]:
        return False

    # Move the pieces
    board[new_row][new_col] = board[start_row][start_col]  # Puller moves
```

```
    board[start_row][start_col] = board[end_row][end_col]  # Pulled piece moves
    board[end_row][end_col] = " "  # Old spot is empty

    move_count += 2
    return True

def check_traps(board = board):
    for trap_row, trap_col in TRAPS:
        piece = board[trap_row][trap_col]
        if piece == " ":
            continue

        # Look around the trap
        has_friend = False
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        for dir_row, dir_col in directions:
            new_row = trap_row + dir_row
            new_col = trap_col + dir_col
            if 0 <= new_row < BOARD_SIZE and 0 <= new_col < BOARD_SIZE:
                nearby_piece = board[new_row][new_col]
                if nearby_piece != " " and nearby_piece[0] == piece[0]:
                    has_friend = True
                    break

        # No friend nearby? Remove the piece
        if not has_friend:
            board[trap_row][trap_col] = " "

def check_winner(board = board):
    global game_finished, whose_turn
    gr_count = 0
    sr_count = 0
    for row in range(BOARD_SIZE):
```

```
    for col in range(BOARD_SIZE):
        if board[row][col] == 'GR':
            gr_count += 1
        if board[row][col] == 'SR':
            sr_count += 1
if gr_count == 0:
    whose_turn = 'Silver'
    print("Silver Wins")
    game_finished = True
    return True

elif sr_count == 0:
    whose_turn = 'Gold'
    print("Gold wins")
    game_finished = True
    return True

# Gold wins if a rabbit reaches row 0
for col in range(BOARD_SIZE):
    if board[0][col] == "GR":
        whose_turn = "Gold"
        print("Gold Wins")
        game_finished = True
        return True
# Silver wins if a rabbit reaches row 7
for col in range(BOARD_SIZE):
    if board[7][col] == "SR":
        whose_turn = "Silver"
        print("Silver Wins")
        game_finished = True
        return True
return False
```

```
def handle_push_pull(start, end, click, board = board):
    sr, sc = start
    er, ec = end
    r, c = click
    success = False
    if((abs(sr-r) + abs(sc-c) < abs(er-r) + abs(ec-c)) and (abs(sr-r)+abs(sc-c)==1)):
        dr = r - sr
        dc = c - sc
        success = pull(sr,sc,er,ec,dr,dc)
    elif((abs(sr-r) + abs(sc-c) > abs(er-r) + abs(ec-c)) and (abs(er-r)+abs(ec-c)==1)):
        dr = r - er
        dc = c - ec
        success = push(sr,sc,er,ec,dr,dc)
    return success


def heuristic(board, add_noise=False):
    h = 0

    # Piece value weights
    piece_values = {
        'SE': 100, 'SC': 50, 'SH': 30, 'SD': 20, 'SCT': 10, 'SR': 10,
        'GE': -100, 'GC': -50, 'GH': -30, 'GD': -20, 'GCT': -10, 'GR': -10,
        ' ': 0
    }

    # Count material
    for row in range(BOARD_SIZE):
        for col in range(BOARD_SIZE):
            piece = board[row][col]
            h += piece_values.get(piece, 0)

    # Rabbit advancement
    for row in range(BOARD_SIZE):
```

```
    for col in range(BOARD_SIZE):
        piece = board[row][col]
        if piece == 'SR':
            h += (row + 1) ** 2
            if row == 7:
                h = float('inf')
            elif row >= 6:
                h += 200
            elif row >= 5:
                h += 100
        elif piece == 'GR':
            h -= (8 - row) ** 2
            if row == 0:
                h = -float('inf')

# Control of center
center_value = [
    [1, 1, 2, 2, 2, 2, 1, 1],
    [1, 2, 3, 3, 3, 3, 2, 1],
    [2, 3, 4, 4, 4, 4, 3, 2],
    [2, 3, 4, 5, 5, 4, 3, 2],
    [2, 3, 4, 5, 5, 4, 3, 2],
    [2, 3, 4, 4, 4, 4, 3, 2],
    [1, 2, 3, 3, 3, 3, 2, 1],
    [1, 1, 2, 2, 2, 2, 1, 1]
]

for row in range(BOARD_SIZE):
    for col in range(BOARD_SIZE):
        piece = board[row][col]
        if piece.startswith('S'):
            h += center_value[row][col] * 2
        elif piece.startswith('G'):
```

```
            h -= center_value[row][col] * 2

    # Trap control
    for trap_row, trap_col in TRAPS:
        silver_adjacent = 0
        gold_adjacent = 0

        for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            r, c = trap_row + dr, trap_col + dc
            if 0 <= r < BOARD_SIZE and 0 <= c < BOARD_SIZE:
                piece = board[r][c]
                if piece.startswith('S'):
                    silver_adjacent += 1
                elif piece.startswith('G'):
                    gold_adjacent += 1

        if silver_adjacent > gold_adjacent:
            h += 15 * (silver_adjacent - gold_adjacent)
        elif gold_adjacent > silver_adjacent:
            h -= 15 * (gold_adjacent - silver_adjacent)

        piece_in_trap = board[trap_row][trap_col]
        if piece_in_trap.startswith('S') and silver_adjacent == 0:
            h -= 50
        elif piece_in_trap.startswith('G') and gold_adjacent == 0:
            h += 50

    # Piece mobility
    silver_mobility = 0
    gold_mobility = 0

    for row in range(BOARD_SIZE):
        for col in range(BOARD_SIZE):
```

```
            piece = board[row][col]
            if piece == " ":
                continue

            moves = 0
            for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                r, c = row + dr, col + dc
                if 0 <= r < BOARD_SIZE and 0 <= c < BOARD_SIZE and board[r][c] == " ":
                    if piece == "SR" and dr == 1:
                        continue
                    if piece == "GR" and dr == -1:
                        continue

                    if not is_frozen(row, col, board):
                        moves += 1

            if piece.startswith('S'):
                silver_mobility += moves
            elif piece.startswith('G'):
                gold_mobility += moves

    h += (silver_mobility - gold_mobility) * 2

    # Elephant positioning
    for row in range(BOARD_SIZE):
        for col in range(BOARD_SIZE):
            if board[row][col] == 'SE':
                center_dist = abs(row - 3.5) + abs(col - 3.5)
                h += (7 - center_dist) * 3

                for dr in range(-2, 3):
                    for dc in range(-2, 3):
                        r, c = row + dr, col + dc
```

```
                if 0 <= r < BOARD_SIZE and 0 <= c < BOARD_SIZE:
                    if board[r][c].startswith('G'):
                        h += 5 / (abs(dr) + abs(dc) + 1)

        elif board[row][col] == 'GE':
            center_dist = abs(row - 3.5) + abs(col - 3.5)
            h -= (7 - center_dist) * 3

# Formation
for row in range(BOARD_SIZE):
    for col in range(BOARD_SIZE):
        piece = board[row][col]
        if piece.startswith('S'):
            friends = 0
            for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                r, c = row + dr, col + dc
                if 0 <= r < BOARD_SIZE and 0 <= c < BOARD_SIZE:
                    if board[r][c].startswith('S'):
                        friends += 1
            h += friends * 2

        elif piece.startswith('G'):
            friends = 0
            for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                r, c = row + dr, col + dc
                if 0 <= r < BOARD_SIZE and 0 <= c < BOARD_SIZE:
                    if board[r][c].startswith('G'):
                        friends += 1
            h -= friends * 2

if add_noise:
    h += random.uniform(-20, 20)
```

```
    return h


def generate_moves(board, current_turn, move_count=0):
    moves = []

    if move_count < 4:
        for row in range(BOARD_SIZE):
            for col in range(BOARD_SIZE):
                piece = board[row][col]
                if piece != " " and piece[0] == current_turn[0]:
                    # Regular moves
                    for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                        new_row = row + dr
                        new_col = col + dc
                        if can_move(row, col, new_row, new_col, board):
                            moves.append((row, col, new_row, new_col, "move"))

                    # Push/pull moves
                    if move_count < 3:
                        for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                            adj_row = row + dr
                            adj_col = col + dc
                            if can_push_or_pull(row, col, adj_row, adj_col, board):
                                # Push directions
                                for pdr, pdc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                                    push_row = adj_row + pdr
                                    push_col = adj_col + pdc
                                    if 0 <= push_row < BOARD_SIZE and 0 <= push_col < BOARD_SIZE:
                                        if board[push_row][push_col] == " ":
                                            moves.append((row, col, adj_row, adj_col, "push", pdr, pdc))

                                # Pull directions
                                for pdr, pdc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
```

```
                    pull_row = row + pdr
                    pull_col = col + pdc
                    if 0 <= pull_row < BOARD_SIZE and 0 <= pull_col < BOARD_SIZE:
                        if board[pull_row][pull_col] == " ":
                            moves.append((row, col, adj_row, adj_col, "pull", pdr, pdc))

        if move_count >= 1:
            moves.append(("pass", None, None, None, None))

    return moves

def make_move(board, move):
    if move[0] == "pass":
        return [row[:] for row in board]

    new_board = [row[:] for row in board]

    if move[4] == "move":
        start_row, start_col, end_row, end_col, _ = move
        new_board[end_row][end_col] = new_board[start_row][start_col]
        new_board[start_row][start_col] = " "

    elif move[4] == "push":
        start_row, start_col, end_row, end_col, _, dir_row, dir_col = move
        push_row, push_col = end_row + dir_row, end_col + dir_col

        new_board[push_row][push_col] = new_board[end_row][end_col]
        new_board[end_row][end_col] = new_board[start_row][start_col]
        new_board[start_row][start_col] = " "

    elif move[4] == "pull":
        start_row, start_col, end_row, end_col, _, dir_row, dir_col = move
        pull_row, pull_col = start_row + dir_row, start_col + dir_col
```

```
        new_board[pull_row][pull_col] = new_board[start_row][start_col]
        new_board[start_row][start_col] = new_board[end_row][end_col]
        new_board[end_row][end_col] = " "

    check_traps(new_board)
    return new_board

def minimax(board, depth, alpha, beta, maximizing_player, current_turn):
    if depth == 0 or check_winner(board):
        # Negate the evaluation for Silver's perspective
        eval_score = heuristic(board, add_noise=(depth == 0))
        if current_turn == "Silver":
            eval_score = -eval_score  # Invert the score for Silver
        return eval_score, None

    moves = generate_moves(board, current_turn)

    if not moves:
        eval_score = heuristic(board)
        if current_turn == "Silver":
            eval_score = -eval_score  # Invert the score for Silver
        return eval_score, None

    random.shuffle(moves)

    if maximizing_player == current_turn:
        max_eval = float('-inf')
        best_move = None

        for move in moves:
            new_board = make_move(board, move)
            eval_score, _ = minimax(new_board, depth - 1, alpha, beta, False, "Gold")
```

```
            if eval_score > max_eval:
                max_eval = eval_score
                best_move = move
            alpha = max(alpha, eval_score)
            if beta <= alpha:
                break
        return max_eval, best_move

    else:
        min_eval = float('inf')
        best_move = None

        for move in moves:
            new_board = make_move(board, move)
            eval_score, _ = minimax(new_board, depth - 1, alpha, beta, True, "Silver")

            if eval_score < min_eval:
                min_eval = eval_score
                best_move = move
            beta = min(beta, eval_score)
            if beta <= alpha:
                break
        return min_eval, best_move

def get_best_move(board, current_turn):
    moves = generate_moves(board, current_turn)

    if len(moves) <= 1:
        return None if len(moves) == 0 else moves[0]

    non_pass_moves = [m for m in moves if m[0] != "pass"]
    if len(non_pass_moves) == 0:
```

```
        return moves[0]

    piece_count = sum(1 for row in board for piece in row if piece != " ")
    depth = 2

    if piece_count < 10:
        depth = 3

    start_time = time.time()
    score, best_move = minimax(board, depth, float('-inf'), float('inf'), current_turn == "Silver",
current_turn)
    end_time = time.time()

    print(f"Minimax search (depth {depth}) took {end_time - start_time:.2f} seconds, score: {score}")

    if best_move is None or best_move[0] == "pass":
        if len(non_pass_moves) > 0:
            print("Minimax defaulting to random non-pass move")
            best_move = random.choice(non_pass_moves)

    return best_move

def handle_ai_turn():
    global whose_turn, move_count, game_finished, board

    remaining_moves = 4 - move_count

    for _ in range(remaining_moves):
        if game_finished:
            break

        ai_move = get_best_move(board, "Silver")
        if ai_move is None:
```

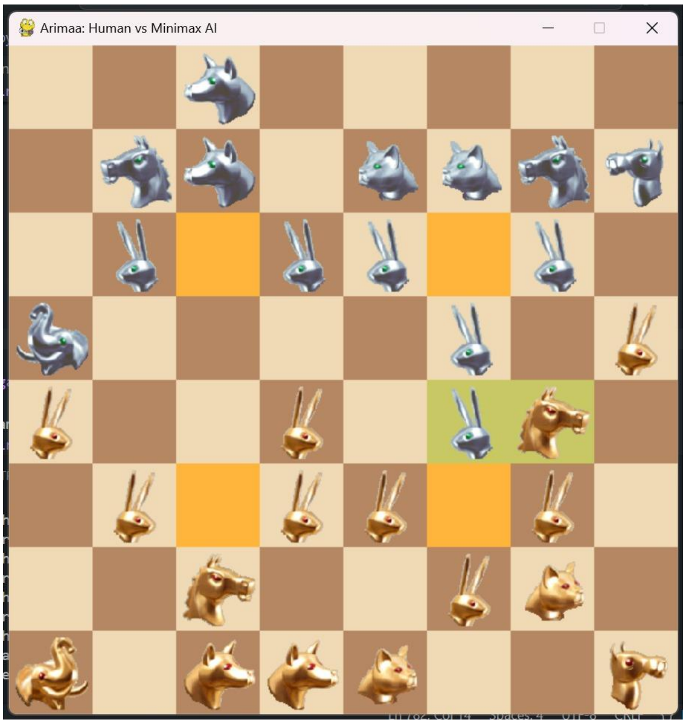
```
            print("AI couldn't find a valid move")
            break

        if ai_move[4] == "move":
            start_row, start_col, end_row, end_col, _ = ai_move
            piece = board[start_row][start_col]
            print(f"AI moves {piece} from {start_row},{start_col} to {end_row},{end_col}")

            board[end_row][end_col] = piece
            board[start_row][start_col] = ' '
            move_count += 1

        elif ai_move[4] == "push":
            start_row, start_col, end_row, end_col, _, dir_row, dir_col = ai_move
            pusher = board[start_row][start_col]
            pushed = board[end_row][end_col]
            push_to_row, push_to_col = end_row + dir_row, end_col + dir_col

            print(f"AI pushes {pushed} at {end_row},{end_col} to {push_to_row},{push_to_col} with
{pusher}")

            board[push_to_row][push_to_col] = pushed
            board[end_row][end_col] = pusher
            board[start_row][start_col] = ' '
            move_count += 2

        elif ai_move[4] == "pull":
            start_row, start_col, end_row, end_col, _, dir_row, dir_col = ai_move
            puller = board[start_row][start_col]
            pulled = board[end_row][end_col]
            puller_to_row, puller_to_col = start_row + dir_row, start_col + dir_col
```

```
            print(f"AI pulls {pulled} from {end_row},{end_col} to {start_row},{start_col} while moving {puller} to {puller_to_row},{puller_to_col}")

            board[puller_to_row][puller_to_col] = puller
            board[start_row][start_col] = pulled
            board[end_row][end_col] = ' '
            move_count += 2

        check_traps()

        if check_winner():
            break

        if move_count >= 4:
            break

    print("AI turn complete")
    whose_turn = "Gold"
    move_count = 0

def handle_click(x, y, button, team):
    global selected, whose_turn, move_count, game_finished

    if whose_turn != team or game_finished:
        return

    col = x // CELL_SIZE
    row = y // CELL_SIZE

    if button == 1:  # Left click (Move / Push / Pull)
        if selected is None:
            # Select piece if it belongs to the current team
            if board[row][col] != " " and board[row][col][0] == team[0]:
```

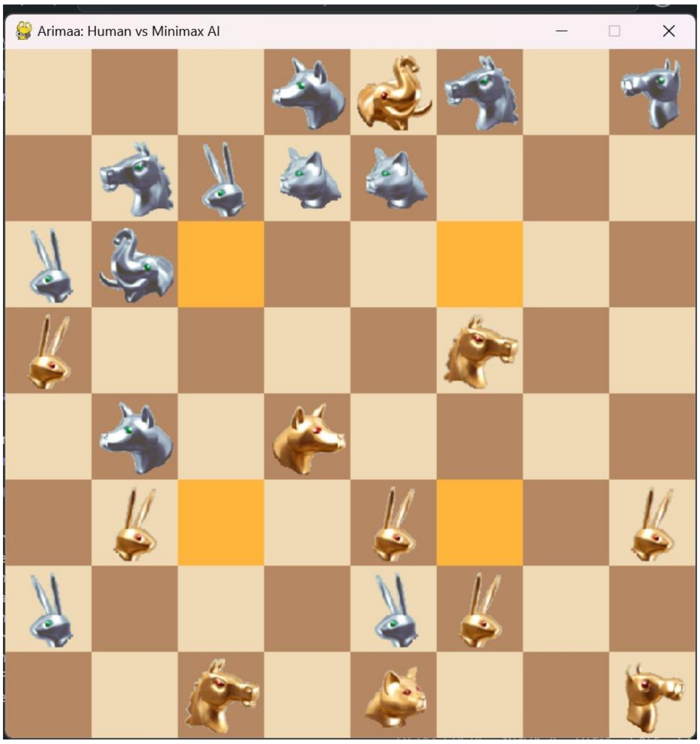
```
        selected = (row, col)
else:
    if isinstance(selected, tuple) and len(selected) == 2 and isinstance(selected[0], int):
        start_row, start_col = selected

        if (row, col) == (start_row, start_col):  # Deselect on re-click
            selected = None
        elif can_move(start_row, start_col, row, col):  # Normal move
            board[row][col], board[start_row][start_col] = board[start_row][start_col], " "
            move_count += 1
            check_traps()
            selected = None
            if check_winner():
                return
            if move_count >= 4:
                whose_turn = "Silver"
                move_count = 0
                handle_ai_turn()
        else:
            selected = None

    elif isinstance(selected, tuple) and len(selected) == 2 and isinstance(selected[0], tuple):
        (start_row, start_col), (end_row, end_col) = selected

        if (row, col) in [(start_row, start_col), (end_row, end_col)]:  # Deselect on re-click
            selected = None
        else:
            dir_row, dir_col = row - end_row, col - end_col
            if can_push_or_pull(start_row, start_col, end_row, end_col) and move_count < 3:
                success = handle_push_pull(selected[0], selected[1], (row, col))
                if success:
                    check_traps()
                    selected = None
```

```
                if check_winner():
                    return
                if move_count >= 4:
                    whose_turn = "Silver"
                    move_count = 0
                    handle_ai_turn()
            else:
                selected = None
        else:
            selected = None

    elif button == 3:  # Right click (Set up push/pull)
        if selected is None:
            if board[row][col] != " " and board[row][col][0] == team[0]:
                selected = (row, col)
        elif isinstance(selected, tuple) and len(selected) == 2 and isinstance(selected[0], int):
            start_row, start_col = selected

            if (row, col) == (start_row, start_col):  # Deselect on re-click
                selected = None
            elif can_push_or_pull(start_row, start_col, row, col):  # Set up push/pull
                selected = ((start_row, start_col), (row, col))
            else:
                selected = None

def pass_turn():
    global whose_turn, move_count, game_finished
    if move_count >= 1 and not game_finished:
        whose_turn = "Silver"
        move_count = 0
        selected = None
        handle_ai_turn()
```

```python
def main():
    running = True
    while running:
        # Clear the screen
        screen.fill((0, 0, 0))
        draw_board()
        pygame.display.flip()

        # Handle events (clicks, key presses)
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
            elif event.type == pygame.MOUSEBUTTONDOWN and not game_finished:
                x, y = event.pos
                button = event.button
                if whose_turn == "Gold":
                    handle_click(x, y, button, "Gold")
            elif event.type == pygame.KEYDOWN and not game_finished:
                if event.key == pygame.K_p:  # Press 'P' to pass
                    pass_turn()

        # If someone won, wait 2 seconds then quit
        if game_finished:
            pygame.display.flip()
            pygame.time.wait(2000)
            running = False

    pygame.quit()

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


Arimaa: Human vs Minimax AI


Arimaa: Human vs Minimax AI
Gold Wins!



3. HUMAN VS AI(HEURISTIC)

Code → import pygame

```
import os
import random

# Set up the game window
WINDOW_WIDTH = 600
WINDOW_HEIGHT = 600
BOARD_SIZE = 8
CELL_SIZE = WINDOW_WIDTH // BOARD_SIZE  # Each square is 75 pixels

# Colors for the board
LIGHT_COLOR = (240, 217, 181)  # Beige
DARK_COLOR = (181, 136, 99)    # Brown
TRAP_COLOR = (255, 180, 60)    # Amber
HIGHLIGHT_COLOR = (200, 200, 100)  # Yellow

# Trap squares where pieces can be captured
TRAPS = [(2, 2), (2, 5), (5, 2), (5, 5)]

# Starting board (8x8 grid)
board = [
    ["SE", "SH", "SD", "SD", "SCT", "SCT", "SH", "SC"],
    ["SR", "SR", "SR", "SR", "SR", "SR", "SR", "SR"],
    [" ", " ", " ", " ", " ", " ", " ", " "],
    [" ", " ", " ", " ", " ", " ", " ", " "],
    [" ", " ", " ", " ", " ", " ", " ", " "],
    [" ", " ", " ", " ", " ", " ", " ", " "],
    ["GR", "GR", "GR", "GR", "GR", "GR", "GR", "GR"],
    ["GE", "GH", "GD", "GD", "GCT", "GCT", "GH", "GC"]
]

PIECE_IMAGES = {
    'GE': 'gold_elephant.png',
    'GC': 'gold_camel.png',
```

```
    'GH': 'gold_horse.png',
    'GD': 'gold_dog.png',
    'GCT': 'gold_cat.png',
    'GR': 'gold_rabbit.png',
    'SE': 'silver_elephant.png',
    'SC': 'silver_camel.png',
    'SH': 'silver_horse.png',
    'SD': 'silver_dog.png',
    'SCT': 'silver_cat.png',
    'SR': 'silver_rabbit.png'
}

# Piece strengths (higher number = stronger piece)
piece_strength = {
    "GE": 5, "GC": 4, "GH": 3, "GD": 2, "GCT": 1, "GR": 0,
    "SE": 5, "SC": 4, "SH": 3, "SD": 2, "SCT": 1, "SR": 0,
    " ": -1  # Empty space
}
def load_images():
    images = {}
    for piece, filename in PIECE_IMAGES.items():
        path = os.path.join(os.getcwd(), filename) #gets the path to the proper image
        images[piece] = pygame.image.load(path)
        images[piece] = pygame.transform.scale(images[piece], (CELL_SIZE - 10, CELL_SIZE - 10))
#scale
    return images

# Start the game
pygame.init()
piece_images = load_images()
screen = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
pygame.display.set_caption("Arimaa")
```

```
# Game variables
selected = None  # What's clicked: None, (row, col), or ((r1, c1), (r2, c2))
whose_turn = "Gold"  # Whose turn it is
move_count = 0  # How many moves made this turn
game_finished = False  # Is the game over?

# Draw the game board
def draw_board():
    # Loop through each square
    for row in range(BOARD_SIZE):
        for col in range(BOARD_SIZE):
            # Set color: light or dark checkerboard pattern
            if (row + col) % 2 == 0:
                color = LIGHT_COLOR
            else:
                color = DARK_COLOR

            # Traps get a special color
            if (row, col) in TRAPS:
                color = TRAP_COLOR

            # Highlight selected squares if game isn't over
            if selected and not game_finished:
                # Single piece selected
                if type(selected) == tuple and len(selected) == 2 and type(selected[0]) == int:
                    if (row, col) == selected:
                        color = HIGHLIGHT_COLOR
                # Two pieces selected for push/pull
                elif type(selected) == tuple and len(selected) == 2 and type(selected[0]) == tuple:
                    if (row, col) == selected[0] or (row, col) == selected[1]:
                        color = HIGHLIGHT_COLOR

            # Draw the square
```

```
            pygame.draw.rect(screen, color, (col * CELL_SIZE, row * CELL_SIZE, CELL_SIZE,
CELL_SIZE))


            # Draw the piece (just text for simplicity)
            piece = board[row][col]
            if piece in piece_images:
                img_rect = piece_images[piece].get_rect(center=(col * CELL_SIZE + CELL_SIZE / 2, row *
CELL_SIZE + CELL_SIZE / 2))
                screen.blit(piece_images[piece], img_rect)
    # Show win message if game is over
    if game_finished:
        font = pygame.font.Font(None, 48)
        win_message = f"{whose_turn} Wins!"
        text = font.render(win_message, True, (255, 255, 255))
        text_pos = text.get_rect(center=(WINDOW_WIDTH // 2, WINDOW_HEIGHT // 2))
        pygame.draw.rect(screen, (0, 0, 0), text_pos.inflate(20, 20))  # Black background
        screen.blit(text, text_pos)


# Check if a piece can't move (frozen by stronger enemy)
def is_frozen(row, col, board):
    piece = board[row][col]
    if piece == " ":
        return False


    frozen = False
    has_friend = False


    # Check all 4 directions: up, down, left, right
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for dir_row, dir_col in directions:
        new_row = row + dir_row
        new_col = col + dir_col
        # Make sure we're still on the board
```

```
        if 0 <= new_row < BOARD_SIZE and 0 <= new_col < BOARD_SIZE:
            nearby_piece = board[new_row][new_col]
            if nearby_piece != " ":
                # Friend nearby (same team)?
                if nearby_piece[0] == piece[0]:
                    has_friend = True
                # Stronger enemy nearby?
                elif piece_strength[piece] < piece_strength[nearby_piece]:
                    frozen = True

    # Frozen only if there's a stronger enemy and no friends
    return frozen and not has_friend

# Check if a move is allowed (one step, not frozen)
def can_move(start_row, start_col, end_row, end_col, board = board):
    # Must be on the board and to an empty space
    if not (0 <= end_row < BOARD_SIZE and 0 <= end_col < BOARD_SIZE):
        return False
    if board[end_row][end_col] != " " or is_frozen(start_row, start_col, board):
        return False

    piece = board[start_row][start_col]
    row_change = start_row - end_row
    col_change = abs(start_col - end_col)

    # Must move exactly one step
    if abs(row_change) + col_change != 1:
        return False

    # Rabbits have special rules
    if piece == "GR":  # Gold rabbit: backward or sideways
        return row_change == 1 or col_change == 1
    if piece == "SR":  # Silver rabbit: forward or sideways
```

```
        return row_change == -1 or col_change == 1
    return True

# Check if push or pull is allowed
def can_push_or_pull(start_row, start_col, end_row, end_col, board = board):
    if 0 <= start_row < 8 and 0 <= start_col < 8 and 0 <= end_row < 8 and 0 <= end_col < 8:
        piece = board[start_row][start_col]
        target = board[end_row][end_col]
    else:
        return False

    # Must be next to each other
    if abs(start_row - end_row) + abs(start_col - end_col) != 1:
        return False

    # Gold pushing/pulling Silver, or Silver pushing/pulling Gold
    if piece[0] == "G" and target[0] == "S" and piece_strength[piece] > piece_strength[target]:
        return True
    if piece[0] == "S" and target[0] == "G" and piece_strength[piece] > piece_strength[target]:
        return True
    return False

# Push a piece (move both pieces)
def push(start_row, start_col, end_row, end_col, dir_row, dir_col, board = board):
    global move_count
    new_row = end_row + dir_row
    new_col = end_col + dir_col

    # Check if the new spot is on the board and empty
    if not (0 <= new_row < BOARD_SIZE and 0 <= new_col < BOARD_SIZE):
        return False  # Can't push out of bounds
    if is_frozen(start_row, start_col, board) or board[new_row][new_col] != " ":
        return False  # Destination must be empty
```

```
    # Ensure the pusher is stronger than the pushed piece
    if piece_strength[board[start_row][start_col]] <= piece_strength[board[end_row][end_col]]:
        return False

    # Move the pushed piece
    board[new_row][new_col] = board[end_row][end_col]
    # Move the pusher to the pushed piece's old spot
    board[end_row][end_col] = board[start_row][start_col]
    # Empty the pusher's original spot
    board[start_row][start_col] = " "

    move_count += 2
    return True

#Pull a piece (move both pieces)
def pull(start_row, start_col, end_row, end_col, dir_row, dir_col, board = board):
    global move_count
    new_row = start_row + dir_row  # The puller moves in the opposite direction of the pull
    new_col = start_col + dir_col

    # Check if the new spot is on the board and empty
    if not (0 <= new_row < BOARD_SIZE and 0 <= new_col < BOARD_SIZE):
        return False
    if is_frozen(start_row, start_col, board) or board[new_row][new_col] != " ":
        return False

    if piece_strength[board[start_row][start_col]] <= piece_strength[board[end_row][end_col]]:
        return False

    # Move the pieces
    board[new_row][new_col] = board[start_row][start_col]  # Puller moves
    board[start_row][start_col] = board[end_row][end_col]  # Pulled piece moves
```

```
    board[end_row][end_col] = " "  # Old spot is empty

    move_count += 2
    return True

# Check if a piece in a trap should be removed
def check_traps(board = board):
    for trap_row, trap_col in TRAPS:
        piece = board[trap_row][trap_col]
        if piece == " ":
            continue

        # Look around the trap
        has_friend = False
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        for dir_row, dir_col in directions:
            new_row = trap_row + dir_row
            new_col = trap_col + dir_col
            if 0 <= new_row < BOARD_SIZE and 0 <= new_col < BOARD_SIZE:
                nearby_piece = board[new_row][new_col]
                if nearby_piece != " " and nearby_piece[0] == piece[0]:
                    has_friend = True
                    print("yay")
                    break

        # No friend nearby? Remove the piece
        if not has_friend:
            print("aww")
            board[trap_row][trap_col] = " "

# Check if someone won (rabbit reached the other side)
def check_winner(board = board):
    global game_finished, whose_turn
```

```
gr_count = 0
sr_count = 0
for row in range(BOARD_SIZE):
    for col in range(BOARD_SIZE):
        if board[row][col] == 'GR':
            gr_count += 1
        if board[row][col] == 'SR':
            sr_count += 1
if gr_count == 0:
    whose_turn = 'Silver'
    game_finished = True
    return True

elif sr_count == 0:
    whose_turn = 'Gold'
    game_finished = True
    return True

# Gold wins if a rabbit reaches row 0
for col in range(BOARD_SIZE):
    if board[0][col] == "GR":
        whose_turn = "Gold"
        game_finished = True
        return True
# Silver wins if a rabbit reaches row 7
for col in range(BOARD_SIZE):
    if board[7][col] == "SR":
        whose_turn = "Silver"
        game_finished = True
        return True
return False
```

```
def handle_push_pull(start, end, click, board = board):
    sr, sc = start
    er, ec = end
    r, c = click
    success = False
    if((abs(sr-r) + abs(sc-c) < abs(er-r) + abs(ec-c)) and (abs(sr-r)+abs(sc-c)==1)):
        dr = r - sr
        dc = c - sc

        success = pull(sr,sc,er,ec,dr,dc)
    elif((abs(sr-r) + abs(sc-c) > abs(er-r) + abs(ec-c)) and (abs(er-r)+abs(ec-c)==1)):
        dr = r - er
        dc = c - ec

        success = push(sr,sc,er,ec,dr,dc)
    return success

def heuristic(board):

    h = 0

    # each piece type has a value
    piece_values = {
        'SE': 100, 'SC': 50, 'SH': 30, 'SD': 20, 'SCT': 10, 'SR': 10,
        'GE': -100, 'GC': -50, 'GH': -30, 'GD': -20, 'GCT': -10, 'GR': -10,
        ' ': 0
    }

    # Count material
    for row in range(BOARD_SIZE):
```

```
    for col in range(BOARD_SIZE):
        piece = board[row][col]
        h += piece_values.get(piece, 0)

# Rabbit advancement - Silver rabbits want to go down, Gold rabbits want to go up
for row in range(BOARD_SIZE):
    for col in range(BOARD_SIZE):
        piece = board[row][col]
        if piece == 'SR':
            # Exponential reward for advancement (7^2=49, 6^2=36, 5^2=25, etc.)
            h += (row + 1) ** 2
            # Extra bonus for being close to the goal row
            if row == 7:
                h = float('inf')
            elif row >= 6:  # One step away from winning
                h += 200
            elif row >= 5:  # Two steps away
                h += 100

        elif piece == 'GR':
            # Penalize Gold rabbit advancement
            h -= (8 - row) ** 2
            if row == 0:
                h = -float('inf')

# Control of center - pieces in the center have more influence
center_value = [
    [1, 1, 2, 2, 2, 2, 1, 1],
    [1, 2, 3, 3, 3, 3, 2, 1],
    [2, 3, 4, 4, 4, 4, 3, 2],
    [2, 3, 4, 5, 5, 4, 3, 2],
    [2, 3, 4, 5, 5, 4, 3, 2],
    [2, 3, 4, 4, 4, 4, 3, 2],
```

```
    [1, 2, 3, 3, 3, 3, 2, 1],
    [1, 1, 2, 2, 2, 2, 1, 1]
]

for row in range(BOARD_SIZE):
    for col in range(BOARD_SIZE):
        piece = board[row][col]
        if piece.startswith('S'):
            h += center_value[row][col] * 2
        elif piece.startswith('G'):
            h -= center_value[row][col] * 2

# Trap control - reward controlling squares adjacent to traps
for trap_row, trap_col in TRAPS:
    silver_adjacent = 0
    gold_adjacent = 0

    for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
        r, c = trap_row + dr, trap_col + dc
        if 0 <= r < BOARD_SIZE and 0 <= c < BOARD_SIZE:
            piece = board[r][c]
            if piece.startswith('S'):
                silver_adjacent += 1
            elif piece.startswith('G'):
                gold_adjacent += 1

    # Reward for trap control (more friendly pieces than enemy pieces)
    if silver_adjacent > gold_adjacent:
        h += 15 * (silver_adjacent - gold_adjacent)
    elif gold_adjacent > silver_adjacent:
        h -= 15 * (gold_adjacent - silver_adjacent)

    # Extra penalty for having a piece in a trap with no friendly support
```

```python
    piece_in_trap = board[trap_row][trap_col]
    if piece_in_trap.startswith('S') and silver_adjacent == 0:
        h -= 50  # Severe penalty for unsupported piece in trap
    elif piece_in_trap.startswith('G') and gold_adjacent == 0:
        h += 50  # Reward for enemy piece about to be captured

# Piece mobility - reward having more available moves
silver_mobility = 0
gold_mobility = 0

for row in range(BOARD_SIZE):
    for col in range(BOARD_SIZE):
        piece = board[row][col]
        if piece == " ":
            continue

        # Count possible moves for this piece
        moves = 0
        for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            r, c = row + dr, col + dc
            if 0 <= r < BOARD_SIZE and 0 <= c < BOARD_SIZE and board[r][c] == " ":
                # Check rabbit movement restrictions
                if piece == "SR" and dr == 1:  # Silver rabbit can't move backward
                    continue
                if piece == "GR" and dr == -1:  # Gold rabbit can't move backward
                    continue

                # Check if piece is frozen
                if not is_frozen(row, col, board):
                    moves += 1

        # Add to team's mobility score
        if piece.startswith('S'):
```

```
            silver_mobility += moves
        elif piece.startswith('G'):
            gold_mobility += moves

# Reward having more mobility than opponent
h += (silver_mobility - gold_mobility) * 2

# Elephant positioning - reward Silver elephant for being in useful positions
for row in range(BOARD_SIZE):
    for col in range(BOARD_SIZE):
        if board[row][col] == 'SE':
            # Reward elephant for being near center
            center_dist = abs(row - 3.5) + abs(col - 3.5)
            h += (7 - center_dist) * 3

            # Reward elephant for being near enemy pieces (to push/pull them)
            for dr in range(-2, 3):
                for dc in range(-2, 3):
                    r, c = row + dr, col + dc
                    if 0 <= r < BOARD_SIZE and 0 <= c < BOARD_SIZE:
                        if board[r][c].startswith('G'):
                            h += 5 / (abs(dr) + abs(dc) + 1)  # Closer is better

        # Penalize Gold elephant for being in useful positions
        elif board[row][col] == 'GE':
            center_dist = abs(row - 3.5) + abs(col - 3.5)
            h -= (7 - center_dist) * 3

# Formation and structure - reward pieces for supporting each other
for row in range(BOARD_SIZE):
    for col in range(BOARD_SIZE):
        piece = board[row][col]
        if piece.startswith('S'):
```

```
            # Count friendly neighbors
            friends = 0
            for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                r, c = row + dr, col + dc
                if 0 <= r < BOARD_SIZE and 0 <= c < BOARD_SIZE:
                    if board[r][c].startswith('S'):
                        friends += 1

            # Reward for having friendly pieces nearby (better formation)
            h += friends * 2

        elif piece.startswith('G'):
            # Count friendly neighbors for Gold
            friends = 0
            for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                r, c = row + dr, col + dc
                if 0 <= r < BOARD_SIZE and 0 <= c < BOARD_SIZE:
                    if board[r][c].startswith('G'):
                        friends += 1

            # Penalize Gold for having good formation
            h -= friends * 2

    return h

def find_best_move(current_board):
    best_move = None
    best_h = heuristic(current_board)
    global move_count

    # Keep track of ANY valid move as a fallback
    fallback_move = None
```

```
# Create a deep copy of the board for simulation
board_copy = [row[:] for row in current_board]

for row in range(BOARD_SIZE):
    for col in range(BOARD_SIZE):
        piece = board_copy[row][col]
        if piece.startswith("S"):
            # Check regular moves
            for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                adj_row = row + dr
                adj_col = col + dc

                if can_move(row, col, adj_row, adj_col, board_copy):
                    # Save the first valid move as a fallback
                    if fallback_move is None:
                        fallback_move = (row, col, adj_row, adj_col, "move")

                    # Test the move
                    test_board = [r[:] for r in board_copy]
                    test_board[adj_row][adj_col] = piece
                    test_board[row][col] = ' '

                    # Simulate trap effects
                    for trap_row, trap_col in TRAPS:
                        trap_piece = test_board[trap_row][trap_col]
                        if trap_piece == " ":
                            continue

                        has_friend = False
                        for tdr, tdc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                            tr, tc = trap_row + tdr, trap_col + tdc
                            if 0 <= tr < BOARD_SIZE and 0 <= tc < BOARD_SIZE:
                                if test_board[tr][tc] != " " and test_board[tr][tc][0] == trap_piece[0]:
```

```
                        has_friend = True
                        break

                if not has_friend:
                    test_board[trap_row][trap_col] = " "

            h = heuristic(test_board)
            if h > best_h:
                best_h = h
                best_move = (row, col, adj_row, adj_col, "move")

    # Check push/pull moves
    for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
        adj_row = row + dr
        adj_col = col + dc

        if move_count < 3 and can_push_or_pull(row, col, adj_row, adj_col, board_copy):
            # Try all possible push directions
            for pdr, pdc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                push_row = adj_row + pdr
                push_col = adj_col + pdc

                if 0 <= push_row < BOARD_SIZE and 0 <= push_col < BOARD_SIZE and
board_copy[push_row][push_col] == " ":
                    # Simulate push
                    test_board = [r[:] for r in board_copy]
                    test_board[push_row][push_col] = test_board[adj_row][adj_col]  # Move pushed
piece
                    test_board[adj_row][adj_col] = test_board[row][col]  # Move pusher
                    test_board[row][col] = " "  # Empty original spot

                    # Simulate trap effects
                    for trap_row, trap_col in TRAPS:
```

```
                    trap_piece = test_board[trap_row][trap_col]
                    if trap_piece == " ":
                        continue

                    has_friend = False
                    for tdr, tdc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                        tr, tc = trap_row + tdr, trap_col + tdc
                        if 0 <= tr < BOARD_SIZE and 0 <= tc < BOARD_SIZE:
                            if test_board[tr][tc] != " " and test_board[tr][tc][0] == trap_piece[0]:
                                has_friend = True
                                break

                    if not has_friend:
                        test_board[trap_row][trap_col] = " "

                h = heuristic(test_board)
                if h > best_h:
                    best_h = h
                    best_move = (row, col, adj_row, adj_col, "push", pdr, pdc)

        # Try all possible pull directions
        for pdr, pdc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            pull_row = row + pdr
            pull_col = col + pdc

            if 0 <= pull_row < BOARD_SIZE and 0 <= pull_col < BOARD_SIZE and
board_copy[pull_row][pull_col] == " ":
                # Simulate pull
                test_board = [r[:] for r in board_copy]
                test_board[pull_row][pull_col] = test_board[row][col]  # Move puller
                test_board[row][col] = test_board[adj_row][adj_col]  # Move pulled piece
                test_board[adj_row][adj_col] = " "  # Empty pulled piece's spot
```

```
                # Simulate trap effects
                for trap_row, trap_col in TRAPS:
                    trap_piece = test_board[trap_row][trap_col]
                    if trap_piece == " ":
                        continue

                    has_friend = False
                    for tdr, tdc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                        tr, tc = trap_row + tdr, trap_col + tdc
                        if 0 <= tr < BOARD_SIZE and 0 <= tc < BOARD_SIZE:
                            if test_board[tr][tc] != " " and test_board[tr][tc][0] == trap_piece[0]:
                                has_friend = True
                                break

                    if not has_friend:
                        test_board[trap_row][trap_col] = " "

                h = heuristic(test_board)
                if h > best_h:
                    best_h = h
                    best_move = (row, col, adj_row, adj_col, "pull", pdr, pdc)

# If we found a better move, return it
if best_move is not None:
    return best_move

# Otherwise, return any valid move
if fallback_move:
    return fallback_move

# If no move is possible (very unlikely), return None
return None
```

```
def handle_ai_turn():
    global whose_turn, move_count, game_finished

    # AI can make up to 4 moves per turn
    remaining_moves = 4 - move_count

    for _ in range(remaining_moves):
        if game_finished:
            break

        ai_move = find_best_move(board)
        if ai_move is None:
            print("AI couldn't find a valid move")
            break

        if ai_move[4] == "move":
            start_row, start_col, end_row, end_col, _ = ai_move
            piece = board[start_row][start_col]
            print(f"AI moves {piece} from {start_row},{start_col} to {end_row},{end_col}")

            # Execute the move
            board[end_row][end_col] = piece
            board[start_row][start_col] = ' '
            move_count += 1

        elif ai_move[4] == "push":
            start_row, start_col, end_row, end_col, _, dir_row, dir_col = ai_move
            pusher = board[start_row][start_col]
            pushed = board[end_row][end_col]
            push_to_row, push_to_col = end_row + dir_row, end_col + dir_col

            print(f"AI pushes {pushed} at {end_row},{end_col} to {push_to_row},{push_to_col} with
{pusher}")
```

```
            # Execute the push
            board[push_to_row][push_to_col] = pushed
            board[end_row][end_col] = pusher
            board[start_row][start_col] = ' '
            move_count += 2

        elif ai_move[4] == "pull":
            start_row, start_col, end_row, end_col, _, dir_row, dir_col = ai_move
            puller = board[start_row][start_col]
            pulled = board[end_row][end_col]
            puller_to_row, puller_to_col = start_row + dir_row, start_col + dir_col

            print(f"AI pulls {pulled} from {end_row},{end_col} to {start_row},{start_col} while moving {puller}
to {puller_to_row},{puller_to_col}")

            # Execute the pull
            board[puller_to_row][puller_to_col] = puller
            board[start_row][start_col] = pulled
            board[end_row][end_col] = ' '
            move_count += 2

        # Check if pieces should be removed from traps
        check_traps()

        # Check if the game is over
        if check_winner():
            break

        # Check if we've used all our moves
        if move_count >= 4:
            break
```

```
    # End AI turn and give control back to player
    print("AI turn complete")
    whose_turn = "Gold"
    move_count = 0

def handle_click(x, y, button, team):
    global selected, whose_turn, move_count, game_finished

    if whose_turn != team or game_finished:
        return

    col = x // CELL_SIZE
    row = y // CELL_SIZE

    if button == 1:  # Left click (Move / Push / Pull)
        if selected is None:
            # Select piece if it belongs to the current team
            if board[row][col] != " " and board[row][col][0] == team[0]:
                selected = (row, col)
        else:
            if isinstance(selected, tuple) and len(selected) == 2 and isinstance(selected[0], int):
                start_row, start_col = selected

                if (row, col) == (start_row, start_col):  # Deselect on re-click
                    selected = None
                elif can_move(start_row, start_col, row, col):  # Normal move
                    board[row][col], board[start_row][start_col] = board[start_row][start_col], " "
                    move_count += 1
                    check_traps()
                    selected = None
                    if check_winner():
                        return
                    if move_count >= 4:
```

```
                whose_turn = "Silver"
                move_count = 0
                # AI's turn after player uses all 4 moves
                handle_ai_turn()
        else:
            selected = None

    elif isinstance(selected, tuple) and len(selected) == 2 and isinstance(selected[0], tuple):
        (start_row, start_col), (end_row, end_col) = selected

        if (row, col) in [(start_row, start_col), (end_row, end_col)]:  # Deselect on re-click
            selected = None
        else:
            dir_row, dir_col = row - end_row, col - end_col
            if can_push_or_pull(start_row, start_col, end_row, end_col) and move_count < 3:
                success = handle_push_pull(selected[0], selected[1], (row, col))
                if success:
                    check_traps()
                    selected = None
                    if check_winner():
                        return
                    if move_count >= 4:
                        whose_turn = "Silver"
                        move_count = 0
                        # AI's turn after player uses all 4 moves
                        handle_ai_turn()
                else:
                    selected = None
            else:
                selected = None

elif button == 3:  # Right click (Set up push/pull)
    if selected is None:
```

```
            if board[row][col] != " " and board[row][col][0] == team[0]:
                selected = (row, col)
        elif isinstance(selected, tuple) and len(selected) == 2 and isinstance(selected[0], int):
            start_row, start_col = selected

            if (row, col) == (start_row, start_col):  # Deselect on re-click
                selected = None
            elif can_push_or_pull(start_row, start_col, row, col):  # Set up push/pull
                selected = ((start_row, start_col), (row, col))
            else:
                selected = None

# Pass the turn to the other player
def pass_turn():
    global whose_turn, move_count, game_finished
    if move_count >= 1 and not game_finished:
        whose_turn = "Silver"
        move_count = 0
        selected = None
        # AI's turn after player passes
        handle_ai_turn()

# Main game loop
def main():
    running = True
    while running:
        # Clear the screen
        screen.fill((0, 0, 0))
        draw_board()
        pygame.display.flip()

        # Handle events (clicks, key presses)
        for event in pygame.event.get():
```

```
            if event.type == pygame.QUIT:
                running = False
            elif event.type == pygame.MOUSEBUTTONDOWN and not game_finished:
                x, y = event.pos
                button = event.button
                if whose_turn == "Gold":
                    handle_click(x, y, button, "Gold")
                # Silver is now handled by AI, so we don't process clicks for Silver
            elif event.type == pygame.KEYDOWN and not game_finished:
                if event.key == pygame.K_p:  # Press 'P' to pass
                    pass_turn()

        # If someone won, wait 2 seconds then quit
        if game_finished:
            pygame.display.flip()
            pygame.time.wait(2000)
            #running = False

    pygame.quit()

# Start the game
if __name__ == "__main__":
    main() # type: ignore
```

Submitted by:

Rahul jilowa → 2301AI18

Chahat Mahajan →2301AI49

Tripti Jain →2301CS60

Chaitanya Sagar →2301CS77

Rincy →2301CS40

Git Hub Link: +https://github.com/tripsycodes/ARIMAA/tree/main