In [9]:
```python
# AI Ludo Game
# Zaid J Adam
```

In [10]:
```python
# Part 1
# Cell 1: Import basic libraries
import numpy as np
import pygame
from   pygame import K_ESCAPE, SCALED, mixer
import random
import time
import torch
import torch.nn as nn
import torch.optim as optim
import os

# Other Useful Modules
import random
import os
from collections import defaultdict

# Machine Learning Modules
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn.ensemble import RandomForestClassifier
from sklearn.utils import resample
```

In [11]:
```python
# Cell 2: Initialize all imported pygame modules
pygame.init()
pygame.display.set_caption("AI Ludo Game") # Set the title of the game window
screen = pygame.display.set_mode((680, 700),SCALED) # Set the display window size t

# Track players as they finish
finished_players = []
```

In [12]:
```python
# Cell 3: name = pygame.image.load('Name.png')
# (The commented line shows an example of how to load a generic image)
board = pygame.image.load(r'C:\Users\Zaid J Adam\Desktop\AI Ludo Dataset and Tools\
star = pygame.image.load(r'C:\Users\Zaid J Adam\Desktop\AI Ludo Dataset and Tools\s
# Load dice face images (1 to 6)
one = pygame.image.load(r'C:\Users\Zaid J Adam\Desktop\AI Ludo Dataset and Tools\1.
two = pygame.image.load(r'C:\Users\Zaid J Adam\Desktop\AI Ludo Dataset and Tools\2.
three = pygame.image.load(r'C:\Users\Zaid J Adam\Desktop\AI Ludo Dataset and Tools\
four = pygame.image.load(r'C:\Users\Zaid J Adam\Desktop\AI Ludo Dataset and Tools\4
five = pygame.image.load(r'C:\Users\Zaid J Adam\Desktop\AI Ludo Dataset and Tools\5
six = pygame.image.load(r'C:\Users\Zaid J Adam\Desktop\AI Ludo Dataset and Tools\6.
# Load images for player colors
red = pygame.image.load(r'C:\Users\Zaid J Adam\Desktop\AI Ludo Dataset and Tools\re
blue = pygame.image.load(r'C:\Users\Zaid J Adam\Desktop\AI Ludo Dataset and Tools\b
green = pygame.image.load(r'C:\Users\Zaid J Adam\Desktop\AI Ludo Dataset and Tools\
yellow = pygame.image.load(r'C:\Users\Zaid J Adam\Desktop\AI Ludo Dataset and Tools

DICE  = [one, two, three, four, five, six] # Group dice face images together
```

```python
color = [red, green, yellow, blue] # Group color images together
# Load sound effects
killSound = mixer.Sound(r'C:\Users\Zaid J Adam\Desktop\AI Ludo Dataset and Tools\Ki
tokenSound = mixer.Sound(r'C:\Users\Zaid J Adam\Desktop\AI Ludo Dataset and Tools\T
diceSound = mixer.Sound(r'C:\Users\Zaid J Adam\Desktop\AI Ludo Dataset and Tools\Di
winnerSound = mixer.Sound(r'C:\Users\Zaid J Adam\Desktop\AI Ludo Dataset and Tools\
```

In [13]:

```python
# Cell 4: Initializing Variables

number        = 1
currentPlayer = 0
playerKilled  = False
diceRolled    = False
winnerRank    = []
start = False
# Font settings
font = pygame.font.Font('freesansbold.ttf', 11)
FONT = pygame.font.Font('freesansbold.ttf', 16)

# Home positions for each player
HOME = [[(110, 58),  (61, 107),  (152, 107), (110, 152)],  # Red
        [(466, 58),  (418, 107), (509, 107), (466, 153)],  # Green
        [(466, 415), (418, 464), (509, 464), (466, 510)],  # Yellow
        [(110, 415), (61, 464),  (152, 464), (110, 510)]]  # Blue

        # Safe positions on the board for each color
SAFE = [(50, 240), (328, 50), (520, 328), (240, 520),
        (88, 328), (240, 88), (482, 240), (328, 482)]
# Positions to place the dice for each player
DicePosition=[(175,173),(531,173),(531,375),(173,375)]
# Main board movement positions
position = [[[110, 58],  [61, 107],  [152, 107], [110, 152]],  # Red
            [[466, 58],  [418, 107], [509, 107], [466, 153]],  # Green
            [[466, 415], [418, 464], [509, 464], [466, 510]],  # Yellow
            [[110, 415], [61, 464],  [152, 464], [110, 510]]]  # Blue
# Jump points: when a player reaches another player's zone
jump = {(202, 240): (240, 202),  # Red to Green
        (328, 202): (368, 240),  # Gren to yellow
        (368, 328): (328, 368),  # Yellow to blue
        (240, 368): (202, 328)}  # Blue to red

# Pre-defined winning path for reaching the center
         # R           G            Y           B
WINNER = [[240, 284], [284, 240], [330, 284], [284, 330]]

# Pre-defined winning path for reaching the center
winner_path = [

        [(12,284),(50, 284), (88, 284), (126, 284), (164, 284), (202, 284), (240, 2
        [(284,12),(284, 50), (284, 88), (284, 126), (284, 164), (284, 202), (284, 2
        [(558,284),(520, 284), (482, 284), (444, 284), (406, 284), (368, 284), (330
        [(284,558),(284, 520), (284, 482), (284, 444), (284, 406), (284, 368), (284
    ]
```

```
# Load default font (used internally by pygame)
pygame.freetype.get_default_font()
```

Out[13]: 'freesansbold.ttf'

In [14]:
```
# Cell 5: Movement
# Function to reset the game state to initial settings
def re_initialize():
    global number,currentPlayer,playerKilled,diceRolled,winnerRank,HOME,SAFE,DicePo
    # Reset game status
    number    = 1
    currentPlayer = 0
    playerKilled  = False
    diceRolled    = False
    winnerRank    = []

    # Reset home positions
    HOME = [[(110, 58),  (61, 107),  (152, 107), (110, 152)],  # Red
            [(466, 58),  (418, 107), (509, 107), (466, 153)],  # Green
            [(466, 415), (418, 464), (509, 464), (466, 510)],  # Yellow
            [(110, 415), (61, 464),  (152, 464), (110, 510)]]  # Blue

    # Reset safe spots on board
            # R          G          Y          B
    SAFE = [(50, 240), (328, 50), (520, 328), (240, 520),
            (88, 328), (240, 88), (482, 240), (328, 482)]
    # Reset dice display positions for each player
    DicePosition=[(175,173),(531,173),(531,375),(173,375)]
    position = [[[110, 58],  [61, 107],  [152, 107], [110, 152]],  # Red
                [[466, 58],  [418, 107], [509, 107], [466, 153]],  # Green
                [[466, 415], [418, 464], [509, 464], [466, 510]],  # Yellow
                [[110, 415], [61, 464],  [152, 464], [110, 510]]]  # Blue

    # Reset jump positions when reaching certain zones
    jump = {(202, 240): (240, 202),  # Red to Green
            (328, 202): (368, 240),  # Gren to yellow
            (368, 328): (328, 368),  # Yellow to blue
            (240, 368): (202, 328)}  # Blue to red

    # Reset winner's path coordinates
            # R           G             Y           B
    WINNER = [[240, 284], [284, 240], [330, 284], [284, 330]]




    # Define constants and global variables
AI_PLAYER_INDEX_R = 0  # AI player 1
AI_PLAYER_INDEX_G = 1  # AI player 2
AI_PLAYER_INDEX_Y = 2  # AI player 3
AI_PLAYER_INDEX_B = 3  # AI player 4
NUM_ACTIONS = 4  # Assuming four tokens per player
```

In [15]:
```
# Cell 6: Define Q-Network
class QNetwork(nn.Module):
```

```
    def __init__(self):
        super(QNetwork, self).__init__()
        self.fc1 = nn.Linear(32, 64)  # Adjusted input size
        self.fc2 = nn.Linear(64, NUM_ACTIONS)

    def forward(self, x):
        x = torch.relu(self.fc1(x)) # Apply ReLU activation after first layer
        x = self.fc2(x) # Output layer without activation (for Q-values)
        return x
```

In [16]:
```
# Cell 7: Initialize Q-Network
r_q_network = QNetwork()
g_q_network = QNetwork()
y_q_network = QNetwork()
b_q_network = QNetwork()
q_network = QNetwork()
```

In [17]:
```
# Cell 8: Load the model if it exists
# Paths to saved model files
r_model_path = 'r_ludo_q_network.pth'
g_model_path = 'g_ludo_q_network.pth'
y_model_path = 'y_ludo_q_network.pth'
b_model_path = 'b_ludo_q_network.pth'
model_path = 'ludo_q_network.pth'

# Load model weights if the files exist
if os.path.exists(r_model_path):
    r_q_network.load_state_dict(torch.load(r_model_path))

if os.path.exists(g_model_path):
    g_q_network.load_state_dict(torch.load(g_model_path))

if os.path.exists(y_model_path):
    y_q_network.load_state_dict(torch.load(y_model_path))

if os.path.exists(b_model_path):
    b_q_network.load_state_dict(torch.load(b_model_path))

if os.path.exists(model_path):
    q_network.load_state_dict(torch.load(model_path))

# Set up optimizers for each Q-network with a learning rate of 0.001
r_optimizer = optim.Adam(r_q_network.parameters(), lr=0.001)
g_optimizer = optim.Adam(g_q_network.parameters(), lr=0.001)
y_optimizer = optim.Adam(y_q_network.parameters(), lr=0.001)
b_optimizer = optim.Adam(b_q_network.parameters(), lr=0.001)
optimizer = optim.Adam(q_network.parameters(), lr=0.001)
```

In [18]:
```
# Cell 9: Define the loss function as Mean Squared Error (MSE)
criterion = nn.MSELoss()
```

In [19]:
```
# Cell 10: Preprocess the game state before feeding into the model
def preprocess_state(state):
    # Flatten the positions into a single list and normalize if necessary
    positions = state['positions']
```

```
    flat_positions = [pos for player in positions for token in player for pos in to
    # Normalize or scale positions if needed
    # ...
    return flat_positions
```

In [20]:
```
# Cell 11: Epsilon-greedy action selection
def choose_action(state, q_network, epsilon=0.1):
    if random.random() < epsilon:
        return random.choice([0, 1, 2, 3])
    else:
        with torch.no_grad():

            state_tensor = torch.tensor(state, dtype=torch.float32)
            q_values = q_network(state_tensor)
            return torch.argmax(q_values).item()
```

In [21]:
```
# Cell 12: Check if all tokens of a player are in the winner rank
def all_in_winner_rank(player):
    # Check if all tokens of the player are in the winner rank
    return all(pos in WINNER[player] for pos in position[player])
```

In [22]:
```
# Cell 13: Check if a specific token has reached the winner rank
def token_reached_winner_rank(player, token_index):
    # Check if the specified token has reached the winner rank
    return position[player][token_index] in WINNER[player]
```

In [23]:
```
# Cell 14: Check if a token is on the winner path
def token_on_winner_path(player, token_index):
    # Implement logic based on your game rules to check if the token is on the path
    # Placeholder logic
    return position[player][token_index] in winner_path[player]
```

In [24]:
```
# Cell 15: Check if a token has reached a safe spot
def token_reached_safe_spot(player, token_index):
    # Check if the specified token has reached a safe spot
    return position[player][token_index] in SAFE
```

In [25]:
```
# Cell 16: Check if a token has left its home
def token_left_home(player, token_index):
    # Check if the specified token has left home
    return position[player][token_index] not in HOME[player]
```

In [26]:
```
# Cell 17: Check if a token has moved forward between two states
def token_moved_forward(old_state, new_state, player, token_index):
    # Check if the specified token has moved forward
    old_pos = old_state['positions'][player][token_index]
    new_pos = new_state['positions'][player][token_index]
    return new_pos > old_pos  # Adjust this logic based on how positions are repres
```

In [27]:
```
# Cell 18: Define reward function for Q-learning
def calculate_reward(old_state, new_state, action_taken, playerKilled, winnerRank,
    reward = 0

    # Assumption: The state includes information about each token's position and st
```

```
    # You need to implement the logic to track this information in your game

    # 1. Getting all your tokens to your winner rank (highest reward)
    if all_in_winner_rank(currentPlayer):
        reward += 100

    # 2. Getting one of your tokens to winner rank
    elif token_reached_winner_rank(currentPlayer, action_taken):
        reward += 50

    # 3. Getting your token to way to winner path
    if token_on_winner_path(currentPlayer, action_taken):
        reward += 25

    # 4. Killing another token
    if playerKilled:
        reward += 20

    # 5. Getting your token to a safe position
    if token_reached_safe_spot(currentPlayer, action_taken):
        reward += 10

    # 6. Getting your token out of home
    if token_left_home(currentPlayer, action_taken):
        reward += 5

    # 7. Moving your token forward (lowest reward)
    if token_moved_forward(old_state, new_state, currentPlayer, action_taken):
        reward += 1

    # Now negative rewards
    # 1. no change in state
    if old_state == new_state:
        reward -= 100

    return reward
```

In [28]:
```
# Cell 19
def get_current_state():
    # This function should return the state as needed by your Q-learning logic.
    # Make sure it aligns with how actions are represented and used.
    # For example, if actions represent moving specific tokens, the state should in
    return [currentPlayer, number, len(position[currentPlayer])]  # Update as neces
```

In [29]:
```
# Cell 20: Function to display tokens and update the game screen
def show_token(x, y):
    screen.fill((0, 0, 0))
    screen.blit(board, (0, 0))
    # screen.blit(name,(0,600))
    for i in SAFE[4:]: # Display stars on safe spots
        screen.blit(star, i)

    # Display all tokens for each playe
    for i in range(len(position)):
        for j in position[i]:
```

```
            screen.blit(color[i], j)

    # Display the current dice face
    screen.blit(DICE[number-1], DicePosition[currentPlayer])

    # Play a sound if the current token reached the winner rank
    if position[x][y] in WINNER:
        winnerSound.play()
    else:
        tokenSound.play()


    # Display the winner rankings on the side
    for i in range(len(winnerRank)):
        rank = FONT.render(f'Position :{i+1}.', True, (0, 0, 0))
        screen.blit(rank, (600, 85 + (40*i)))
        screen.blit(color[winnerRank[i]], (620, 75 + (40*i)))

    pygame.display.update() # Refresh the screen
    time.sleep(0.05) # Short delay to control animation speed
```

In [30]:
```
# Cell 21
def show_all():

    for i in SAFE[4:]:
        screen.blit(star, i)

    for i in range(len(position)):
        for j in position[i]:
            screen.blit(color[i], j)

    screen.blit(DICE[number-1], DicePosition[currentPlayer])



    for i in range(len(winnerRank)):
        rank = FONT.render(f'{i+1}.', True, (0, 0, 0))
        screen.blit(rank, (600, 85 + (40*i)))
        screen.blit(color[winnerRank[i]], (620, 75 + (40*i)))
    # screen.blit(name,(0,600))
```

In [31]:
```
# Cell 22
def is_possible(x, y):
    # Check if token is already on winner position
    if position[x][y] in WINNER:
        return False

    #  R2
    if (position[x][y][1] == 284 and position[x][y][0] <= 202 and x == 0) \
            and (position[x][y][0] + 38*number > WINNER[x][0]):
        return False

    #  Y2
    elif (position[x][y][1] == 284 and 368 < position[x][y][0] and x == 2) \
            and (position[x][y][0] - 38*number < WINNER[x][0]):
```

```
        return False
    #  G2
    elif (position[x][y][0] == 284 and position[x][y][1] <= 202 and x == 1) \
            and (position[x][y][1] + 38*number > WINNER[x][1]):
        return False
    #  B2
    elif (position[x][y][0] == 284 and position[x][y][1] >= 368 and x == 3) \
            and (position[x][y][1] - 38*number < WINNER[x][1]):
        return False
    return True
```

In [32]:

```
# Cell 23
def move_token(x, y):
    global currentPlayer, diceRolled

    # Taking Token out of HOME
    if tuple(position[x][y]) in HOME[currentPlayer] and number == 6:
        position[x][y] = list(SAFE[currentPlayer])
        tokenSound.play()
        diceRolled = False

    # Moving token which is not in HOME
    elif tuple(position[x][y]) not in HOME[currentPlayer]:
        diceRolled = False
        if not number == 6:
            currentPlayer = (currentPlayer+1) % 4

        for i in range(number):
        #     if position[x][y] in winner_path[x]:
        #         position[x][y] = list(winner_path[x][winner_path[x].index(positio
            #  R2
            if (position[x][y][1] == 284 and position[x][y][0] <= 202 and x == 0) \
                    and (position[x][y][0] + 38 <= WINNER[x][0]):
                position[x][y][0] += 38
                show_token(x, y)

            #  Y2
            elif (position[x][y][1] == 284 and 368 < position[x][y][0] and x == 2)
                    and (position[x][y][0] - 38*number >= WINNER[x][0]):
                # for i in range(number):
                position[x][y][0] -= 38
                show_token(x,y)

            #  G2
            elif (position[x][y][0] == 284 and position[x][y][1] <= 202 and x == 1)
                    and (position[x][y][1] + 38*number <= WINNER[x][1]):
                # for i in range(number):
                position[x][y][1] += 38
                show_token(x,y)
            #  B2
            elif (position[x][y][0] == 284 and position[x][y][1] >= 368 and x == 3)
                    and (position[x][y][1] - 38*number >= WINNER[x][1]):
                # for i in range(number):
                position[x][y][1] -= 38
                show_token(x,y)
```

```python
        # Other Paths
            else:
                #  R1, Y3
                if (position[x][y][1] == 240 and position[x][y][0] < 202) \
                        or (position[x][y][1] == 240 and 368 <= position[x][y][0] <
                    position[x][y][0] += 38
                # R3 -> R2 -> R1
                elif (position[x][y][0] == 12 and position[x][y][1] > 240):
                    position[x][y][1] -= 44

                #  R3, Y1
                elif (position[x][y][1] == 328 and 12 < position[x][y][0] <= 202) \
                        or (position[x][y][1] == 328 and 368 < position[x][y][0]):
                    position[x][y][0] -= 38
                #  Y3 -> Y2 -> Y1
                elif (position[x][y][0] == 558 and position[x][y][1] < 328):
                    position[x][y][1] += 44

                #  G3, B1
                elif (position[x][y][0] == 240 and 12 < position[x][y][1] <= 202) \
                        or (position[x][y][0] == 240 and 368 < position[x][y][1]):
                    position[x][y][1] -= 38
                # G3 -> G2 -> G1
                elif (position[x][y][1] == 12 and 240 <= position[x][y][0] < 328):
                    position[x][y][0] += 44

                #  B3, G1
                elif (position[x][y][0] == 328 and position[x][y][1] < 202) \
                        or (position[x][y][0] == 328 and 368 <= position[x][y][1] <
                    position[x][y][1] += 38
                #  B3 -> B2 -> B1
                elif (position[x][y][1] == 558 and position[x][y][0] > 240):
                    position[x][y][0] -= 44

                else:
                    for i in jump:
                        if position[x][y] == list(i):
                            position[x][y] = list(jump[i])
                            break

                show_token(x, y)

        # Ki Player
        if tuple(position[x][y]) not in SAFE:
            for i in range(len(position)):
                for j in range(len(position[i])):
                    if position[i][j] == position[x][y] and i != x:
                        position[i][j] = list(HOME[i][j])
                        killSound.play()
                        currentPlayer = x # (currentPlayer+3) % 4
    check_winner()
```

In [33]:
```python
# Cell 24
def check_winner():
    if len(finished_players) == 4:
        font = pygame.font.Font(None, 36)
```

```
        screen.fill((0, 0, 0))
        for idx, player in enumerate(finished_players):
            text = f"{idx+1} - Player {player + 1} {'(Lost)' if idx == 3 else ''}"
            render = font.render(text, True, (255, 255, 255))  # 'True' is a flag f
            screen.blit(render, (50, 100 + idx * 50))
        pygame.display.update()
        time.sleep(10)
        pygame.quit()
        exit()

    global currentPlayer
    if currentPlayer not in winnerRank:
        for i in position[currentPlayer]:
            if i not in WINNER:
                return
        winnerRank.append(currentPlayer)
    else:
        currentPlayer = (currentPlayer + 1) % 4
```

In [34]:
```
# Cell 25
old_winner_rank = winnerRank.copy()
```

In [35]:
```
# Cell 26: Get the type of gameplay
while True:
    try:
        game_type = int(input("Enter the type of gameplay (1 for all AI players, 2
        if game_type in [1, 2, 3, 4]:
            break
    except ValueError:
        continue

# Main LOOP
running = True
while running:
    screen.fill((0, 0, 0))
    screen.blit(board, (0, 0))  # Blitting Board

    check_winner()

    if len(winnerRank) >= 3:
        running = False
        break

    for event in pygame.event.get():
        if event.type == pygame.QUIT or (event.type == pygame.KEYUP and event.key =
            running = False
            break

        if event.type == pygame.MOUSEBUTTONUP and game_type != 1:
            coordinate = pygame.mouse.get_pos()

            human_turn = False
            if game_type == 2 and currentPlayer == 0:
                human_turn = True
            elif game_type == 3 and currentPlayer in [0, 1]:
```

```
                human_turn = True
            elif game_type == 4 and currentPlayer in [0, 1, 2]:
                human_turn = True

            if human_turn:
                if not diceRolled and (DicePosition[currentPlayer][1] <= coordinate
                    number = random.randint(1, 6)
                    diceSound.play()
                    flag = True
                    for i in range(len(position[currentPlayer])):
                        if tuple(position[currentPlayer][i]) not in HOME[currentPla
                            flag = False
                    if (flag and number == 6) or not flag:
                        diceRolled = True
                    else:
                        currentPlayer = (currentPlayer + 1) % 4
                elif diceRolled:
                    for j in range(len(position[currentPlayer])):
                        if position[currentPlayer][j][0] <= coordinate[0] <= positi
                            move_token(currentPlayer, j)
                            break

    # AI or CPU Move
    ai_turn = False
    if (game_type == 1) or \
       (game_type == 2 and currentPlayer != 0) or \
       (game_type == 3 and currentPlayer not in [0, 1]) or \
       (game_type == 4 and currentPlayer not in [0, 1, 2]):
        ai_turn = True

    if ai_turn:
        if not diceRolled:
            number = random.randint(1, 6)
            diceSound.play()

            flag = True
            for i in range(len(position[currentPlayer])):
                if tuple(position[currentPlayer][i]) not in HOME[currentPlayer] and
                    flag = False
            if (flag and number == 6) or not flag:
                diceRolled = True
            else:
                currentPlayer = (currentPlayer + 1) % 4

        elif diceRolled:
            old_state = {'positions': [list(player) for player in position]}
            old_state_processed = preprocess_state(old_state)

            if currentPlayer == 0:
                q_network = r_q_network
                optimizer = r_optimizer
            elif currentPlayer == 1:
                q_network = g_q_network
                optimizer = g_optimizer
            elif currentPlayer == 2:
                q_network = y_q_network
```

```
                optimizer = y_optimizer
            else:
                q_network = b_q_network
                optimizer = b_optimizer

            action = choose_action(old_state_processed, q_network)

            retry = 0
            max_retries = 10
            while retry < max_retries:
                if ((tuple(position[currentPlayer][action]) in HOME[currentPlayer]
                    not is_possible(currentPlayer, action) or
                    position[currentPlayer][action] in WINNER or
                    (position[currentPlayer][action] in winner_path[currentPlayer]

                    reward = -10
                    q_values = q_network(torch.tensor(old_state_processed, dtype=to
                    next_q_values = q_network(torch.tensor(old_state_processed, dty
                    max_next_q = torch.max(next_q_values).item()
                    target_q = q_values.clone()
                    target_q[0][action] = reward + 0.9 * max_next_q
                    loss = criterion(q_values, target_q)
                    optimizer.zero_grad()
                    loss.backward()
                    optimizer.step()

                    retry += 1
                    pygame.time.delay(10)
                    action = choose_action(old_state_processed, q_network)
                else:
                    break

            move_token(currentPlayer, action)

            new_state = {'positions': [list(player) for player in position]}
            new_state_processed = preprocess_state(new_state)

            reward = calculate_reward(old_state, new_state, action, playerKilled, w

            q_values = q_network(torch.tensor(old_state_processed, dtype=torch.floa
            next_q_values = q_network(torch.tensor(new_state_processed, dtype=torch
            max_next_q = torch.max(next_q_values).item()
            target_q = q_values.clone()
            target_q[0][action] = reward + 0.9 * max_next_q

            loss = criterion(q_values, target_q)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

pygame.quit()
```

In [36]:
```
# Cell 27: Display Final Rankings

player_names = ["Red", "Green", "Yellow", "Blue"]
```

```
# Prepare winner announcement
for i, player in enumerate(winnerRank):
    print(f"{i+1}st Place: {player_names[player]} Player")

# Check if any player lost
losers = [i for i in range(4) if i not in winnerRank]
for loser in losers:
    print(f"Lost: {player_names[loser]} Player")
```

```
1st Place: Red Player
2st Place: Green Player
3st Place: Blue Player
Lost: Yellow Player
```

In [37]:

```
# Cell 28
import csv

# Initialize a list to store moves
moves_log = []

def log_move(player, piece, from_pos, to_pos, dice_roll):
    moves_log.append({
        'Player': player,
        'Piece': piece,
        'From': from_pos,
        'To': to_pos,
        'Dice Roll': dice_roll
    })

def save_moves_to_csv(filename='game_moves_log.csv'):
    keys = moves_log[0].keys() if moves_log else []
    with open(filename, 'w', newline='') as output_file:
        dict_writer = csv.DictWriter(output_file, keys)
        dict_writer.writeheader()
        dict_writer.writerows(moves_log)
```

In [38]:

```
# Part 2
# Cell 29: Library Imports
# Standard Data Science Libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Game Development Library
import pygame

# Other Useful Modules
import random
import os
from collections import defaultdict

# Machine Learning Modules
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

```
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn.ensemble import RandomForestClassifier
from sklearn.utils import resample
```

In [39]:

```
# Cell 30: Load Dataset
file_path = r"C:\Users\Zaid J Adam\Desktop\AI Ludo Dataset and Tools\ludo_ai_datase
df = pd.read_csv(file_path)
df.head()
```

Out[39]:

	player_id	dice_roll	token1_pos	token2_pos	token3_pos	token4_pos	opponent1_token
0	3	1	48	42	25	45	
1	4	4	30	53	55	18	
2	1	6	36	4	20	9	
3	3	2	48	2	48	35	
4	3	2	54	10	29	41	

5 rows × 31 columns

In [40]:

```
# Cell 31: Data Exploratory Analysis
print("Shape:", df.shape)
print("Types:\n", df.dtypes)
print("Missing:\n", df.isnull().sum())
df.describe(include='all')
```

```
Shape: (100000, 31)
Types:
 player_id                          int64
dice_roll                          int64
token1_pos                         int64
token2_pos                         int64
token3_pos                         int64
token4_pos                         int64
opponent1_token1_pos               int64
opponent1_token2_pos               int64
opponent1_token3_pos               int64
opponent1_token4_pos               int64
opponent2_token1_pos               int64
opponent2_token2_pos               int64
opponent2_token3_pos               int64
opponent2_token4_pos               int64
opponent3_token1_pos               int64
opponent3_token2_pos               int64
opponent3_token3_pos               int64
opponent3_token4_pos               int64
move_choice                        int64
reward                             int64
action                            object
player_pieces_finished             int64
player_pieces_home                 int64
opponent_pieces_home               int64
player_pieces_safe                 int64
player_pieces_ready_to_exit_home   int64
player_pieces_close_to_finish      int64
opponent_pieces_close_to_you       int64
total_pieces_finished              int64
can_any_move                       int64
strategy                          object
dtype: object
Missing:
 player_id                          0
dice_roll                          0
token1_pos                         0
token2_pos                         0
token3_pos                         0
token4_pos                         0
opponent1_token1_pos               0
opponent1_token2_pos               0
opponent1_token3_pos               0
opponent1_token4_pos               0
opponent2_token1_pos               0
opponent2_token2_pos               0
opponent2_token3_pos               0
opponent2_token4_pos               0
opponent3_token1_pos               0
opponent3_token2_pos               0
opponent3_token3_pos               0
opponent3_token4_pos               0
move_choice                        0
reward                             0
action                             0
```

```
player_pieces_finished            0
player_pieces_home                0
opponent_pieces_home              0
player_pieces_safe                0
player_pieces_ready_to_exit_home  0
player_pieces_close_to_finish     0
opponent_pieces_close_to_you      0
total_pieces_finished             0
can_any_move                      0
strategy                          0
dtype: int64
```

Out[40]:

	player_id	dice_roll	token1_pos	token2_pos	token3_pos	toke
count	100000.000000	100000.000000	100000.000000	100000.00000	100000.000000	100000.
unique	NaN	NaN	NaN	NaN	NaN	
top	NaN	NaN	NaN	NaN	NaN	
freq	NaN	NaN	NaN	NaN	NaN	
mean	2.497830	3.496910	28.561940	28.49398	28.474460	28.
std	1.118225	1.707589	16.726673	16.73962	16.751802	16.
min	1.000000	1.000000	0.000000	0.00000	0.000000	0.
25%	1.000000	2.000000	14.000000	14.00000	14.000000	14.
50%	2.000000	3.000000	29.000000	29.00000	28.000000	29.
75%	3.000000	5.000000	43.000000	43.00000	43.000000	43.
max	4.000000	6.000000	57.000000	57.00000	57.000000	57.

11 rows × 31 columns

In [41]:

```
# Cell 32: Feature Description (Manual Summary)
feature_summary = {
    'dice_roll': 'Continuous numeric value (1-6)',
    'player_pieces_home': 'Integer count of player pieces still in base',
    'player_pieces_finished': 'Integer count of player pieces that reached home',
    'opponent_pieces_home': 'Same as player, but for opponent',
    'opponent_pieces_finished': 'Same as player, but for opponent',
    'strategy': 'Categorical: strategy selected (e.g., defensive, aggressive)',
    'action': 'Categorical: action taken based on strategy'
}
for k, v in feature_summary.items():
    print(f"{k}: {v}")
```

```
dice_roll: Continuous numeric value (1-6)
player_pieces_home: Integer count of player pieces still in base
player_pieces_finished: Integer count of player pieces that reached home
opponent_pieces_home: Same as player, but for opponent
opponent_pieces_finished: Same as player, but for opponent
strategy: Categorical: strategy selected (e.g., defensive, aggressive)
action: Categorical: action taken based on strategy
```

In [42]:
```python
# Cell 33: Skewness Check
# First print available columns
print(df.columns.tolist())

# Then do skewness check safely
selected_cols = [col for col in ['dice_roll', 'player_pieces_home', 'player_pieces_
                                 'opponent_pieces_home', 'opponent_pieces_finished'

if selected_cols:
    print("\nSkewness:\n", df[selected_cols].skew())
else:
    print("\nNo matching columns found for skewness check.")
```

```
['player_id', 'dice_roll', 'token1_pos', 'token2_pos', 'token3_pos', 'token4_pos',
'opponent1_token1_pos', 'opponent1_token2_pos', 'opponent1_token3_pos', 'opponent1_t
oken4_pos', 'opponent2_token1_pos', 'opponent2_token2_pos', 'opponent2_token3_pos',
'opponent2_token4_pos', 'opponent3_token1_pos', 'opponent3_token2_pos', 'opponent3_t
oken3_pos', 'opponent3_token4_pos', 'move_choice', 'reward', 'action', 'player_piece
s_finished', 'player_pieces_home', 'opponent_pieces_home', 'player_pieces_safe', 'pl
ayer_pieces_ready_to_exit_home', 'player_pieces_close_to_finish', 'opponent_pieces_c
lose_to_you', 'total_pieces_finished', 'can_any_move', 'strategy']

Skewness:
 dice_roll                 0.001237
player_pieces_home        0.000000
player_pieces_finished    3.691230
opponent_pieces_home      0.000000
dtype: float64
```

In [43]:
```python
# Cell 34: Outlier Detection using Z-Score
from scipy.stats import zscore

# Select only columns that exist
selected_cols = [col for col in ['dice_roll', 'player_pieces_home', 'player_pieces_
                                 'opponent_pieces_home', 'opponent_pieces_finished'

if selected_cols:
    z_scores = np.abs(zscore(df[selected_cols]))
    outliers = (z_scores > 3).any(axis=1)
    print("\nOutliers found:", outliers.sum())
else:
    print("\nNo matching columns found for outlier detection.")
```

```
Outliers found: 6754
```

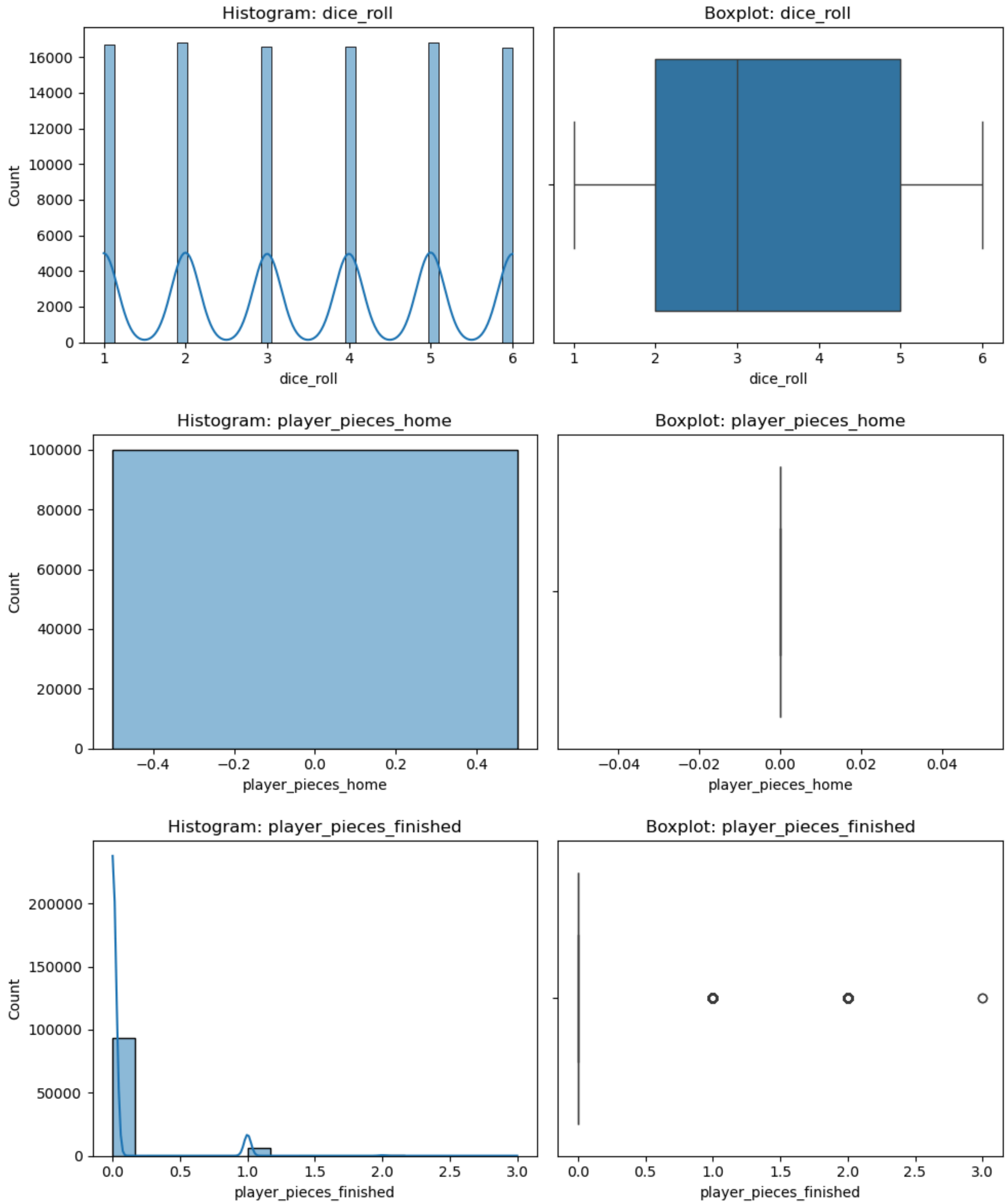
In [44]:
```python
# Cell 35: Class Imbalance Check
if 'action' in df.columns:
    print("\nClass Distribution (action):\n", df['action'].value_counts())
```

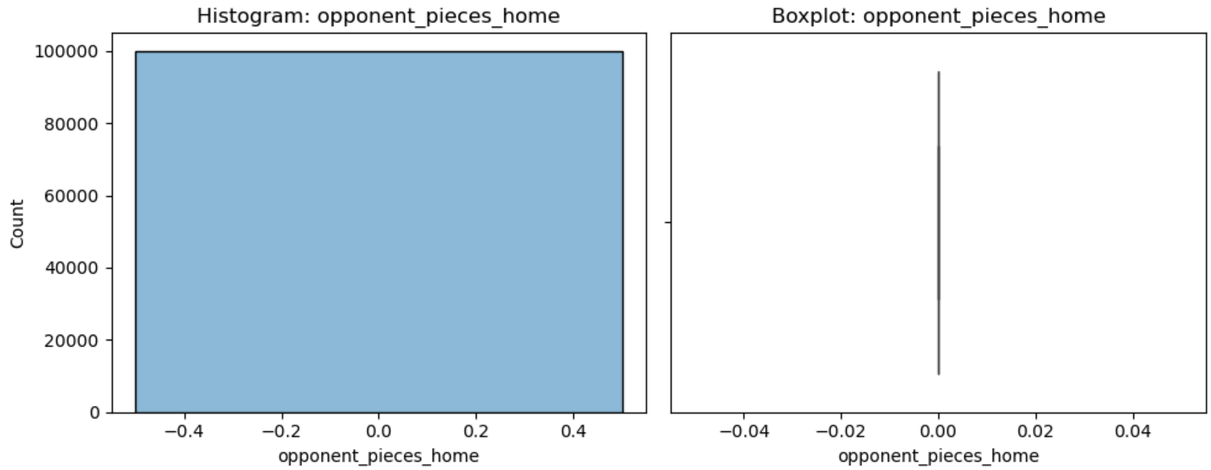
```
else:
    print("\n'action' column not found in the dataset.")
```

```
Class Distribution (action):
 action
Move Token 3 based on dice roll 5    4294
Move Token 1 based on dice roll 2    4293
Move Token 2 based on dice roll 1    4242
Move Token 4 based on dice roll 5    4203
Move Token 4 based on dice roll 1    4200
Move Token 2 based on dice roll 5    4188
Move Token 4 based on dice roll 2    4182
Move Token 1 based on dice roll 6    4181
Move Token 2 based on dice roll 2    4180
Move Token 3 based on dice roll 4    4179
Move Token 4 based on dice roll 4    4169
Move Token 4 based on dice roll 3    4168
Move Token 1 based on dice roll 1    4166
Move Token 2 based on dice roll 3    4153
Move Token 3 based on dice roll 6    4151
Move Token 1 based on dice roll 3    4149
Move Token 3 based on dice roll 2    4141
Move Token 1 based on dice roll 4    4140
Move Token 1 based on dice roll 5    4136
Move Token 4 based on dice roll 6    4117
Move Token 3 based on dice roll 3    4108
Move Token 2 based on dice roll 6    4094
Move Token 2 based on dice roll 4    4092
Move Token 3 based on dice roll 1    4074
Name: count, dtype: int64
```

In [45]:

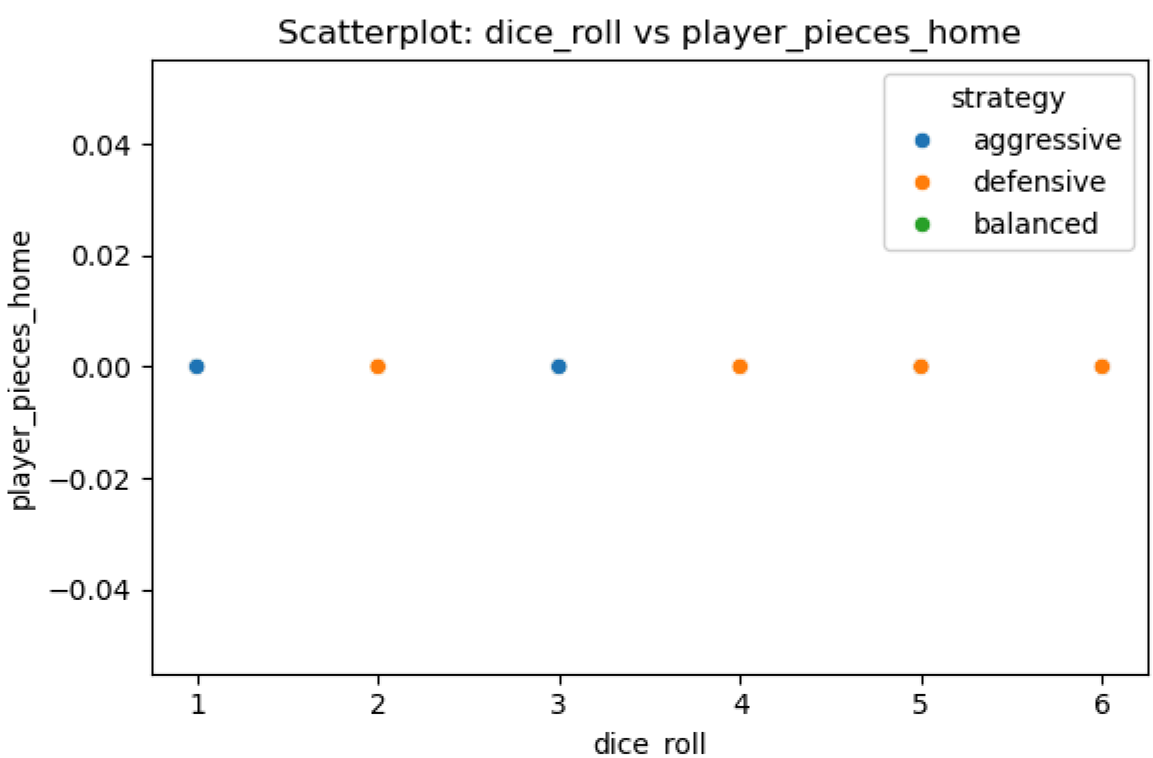
```
# Cell 36: Histograms and Boxplots
numerical = ['dice_roll', 'player_pieces_home', 'player_pieces_finished', 'opponent

# Filter to only existing columns
numerical_existing = [col for col in numerical if col in df.columns]

for col in numerical_existing:
    plt.figure(figsize=(10, 4))

    plt.subplot(1, 2, 1)
    sns.histplot(df[col], kde=True)
    plt.title(f"Histogram: {col}")

    plt.subplot(1, 2, 2)
    sns.boxplot(x=df[col])
    plt.title(f"Boxplot: {col}")

    plt.tight_layout()
    plt.show()
```

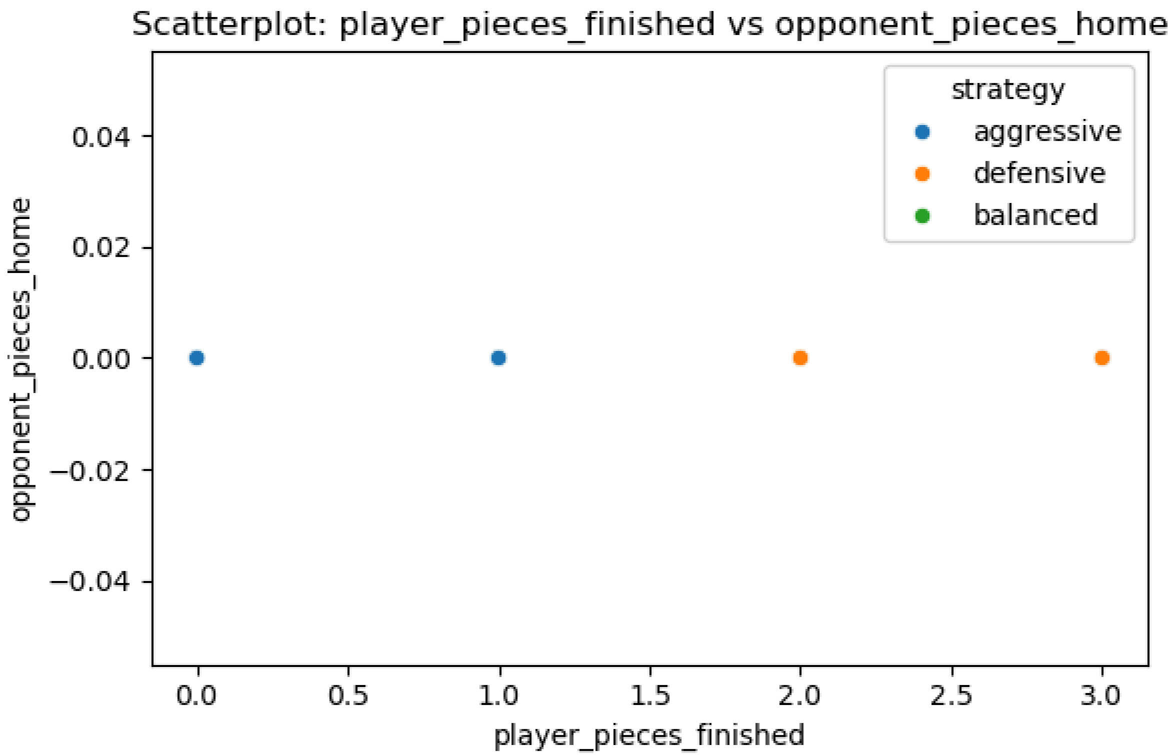

Histogram: dice_roll
Boxplot: dice_roll
Count
dice_roll
dice_roll
Histogram: player_pieces_home
Boxplot: player_pieces_home
Count
player_pieces_home
player_pieces_home
Histogram: player_pieces_finished
Boxplot: player_pieces_finished
Count
player_pieces_finished
player_pieces_finished



In [46]:

```
# Cell 37: Scatterplots for Feature Comparisons
scatter_pairs = [('dice_roll', 'player_pieces_home'),
                 ('player_pieces_finished', 'opponent_pieces_home'),
                 ('opponent_pieces_finished', 'dice_roll')]

# Only keep pairs where both x and y exist
scatter_pairs_existing = [(x, y) for (x, y) in scatter_pairs if x in df.columns and

for x, y in scatter_pairs_existing:
    plt.figure(figsize=(6, 4))
    sns.scatterplot(x=df[x], y=df[y], hue=df['strategy'] if 'strategy' in df.column
    plt.title(f"Scatterplot: {x} vs {y}")
    plt.tight_layout()
    plt.show()
```





In [47]:

```
# Cell 38: Data Augmentation (Upsample minority classes)
# Example augmentation: balancing 'action' using resampling
max_count = df['action'].value_counts().max()
df_balanced = pd.DataFrame()
for action_label in df['action'].unique():
    df_class = df[df['action'] == action_label]
    df_upsampled = resample(df_class, replace=True, n_samples=max_count, random_sta
    df_balanced = pd.concat([df_balanced, df_upsampled])

print("Original dataset shape:", df['action'].value_counts().to_dict())
print("Balanced dataset shape:", df_balanced['action'].value_counts().to_dict())

# Replace original df with balanced version for model training
df = df_balanced.reset_index(drop=True)
```

```
Original dataset shape: {'Move Token 3 based on dice roll 5': 4294, 'Move Token 1 based on dice roll 2': 4293, 'Move Token 2 based on dice roll 1': 4242, 'Move Token 4 based on dice roll 5': 4203, 'Move Token 4 based on dice roll 1': 4200, 'Move Token 2 based on dice roll 5': 4188, 'Move Token 4 based on dice roll 2': 4182, 'Move Token 1 based on dice roll 6': 4181, 'Move Token 2 based on dice roll 2': 4180, 'Move Token 3 based on dice roll 4': 4179, 'Move Token 4 based on dice roll 4': 4169, 'Move Token 4 based on dice roll 3': 4168, 'Move Token 1 based on dice roll 1': 4166, 'Move Token 2 based on dice roll 3': 4153, 'Move Token 3 based on dice roll 6': 4151, 'Move Token 1 based on dice roll 3': 4149, 'Move Token 3 based on dice roll 2': 4141, 'Move Token 1 based on dice roll 4': 4140, 'Move Token 1 based on dice roll 5': 4136, 'Move Token 4 based on dice roll 6': 4117, 'Move Token 3 based on dice roll 3': 4108, 'Move Token 2 based on dice roll 6': 4094, 'Move Token 2 based on dice roll 4': 4092, 'Move Token 3 based on dice roll 1': 4074}
Balanced dataset shape: {'Move Token 4 based on dice roll 1': 4294, 'Move Token 4 based on dice roll 4': 4294, 'Move Token 1 based on dice roll 6': 4294, 'Move Token 3 based on dice roll 3': 4294, 'Move Token 1 based on dice roll 5': 4294, 'Move Token 3 based on dice roll 6': 4294, 'Move Token 1 based on dice roll 1': 4294, 'Move Token 2 based on dice roll 6': 4294, 'Move Token 2 based on dice roll 5': 4294, 'Move Token 1 based on dice roll 3': 4294, 'Move Token 2 based on dice roll 4': 4294, 'Move Token 4 based on dice roll 3': 4294, 'Move Token 4 based on dice roll 2': 4294, 'Move Token 2 based on dice roll 3': 4294, 'Move Token 1 based on dice roll 4': 4294, 'Move Token 3 based on dice roll 2': 4294, 'Move Token 3 based on dice roll 5': 4294, 'Move Token 3 based on dice roll 4': 4294, 'Move Token 2 based on dice roll 1': 4294, 'Move Token 3 based on dice roll 1': 4294, 'Move Token 1 based on dice roll 2': 4294, 'Move Token 2 based on dice roll 2': 4294, 'Move Token 4 based on dice roll 6': 4294, 'Move Token 4 based on dice roll 5': 4294}
```

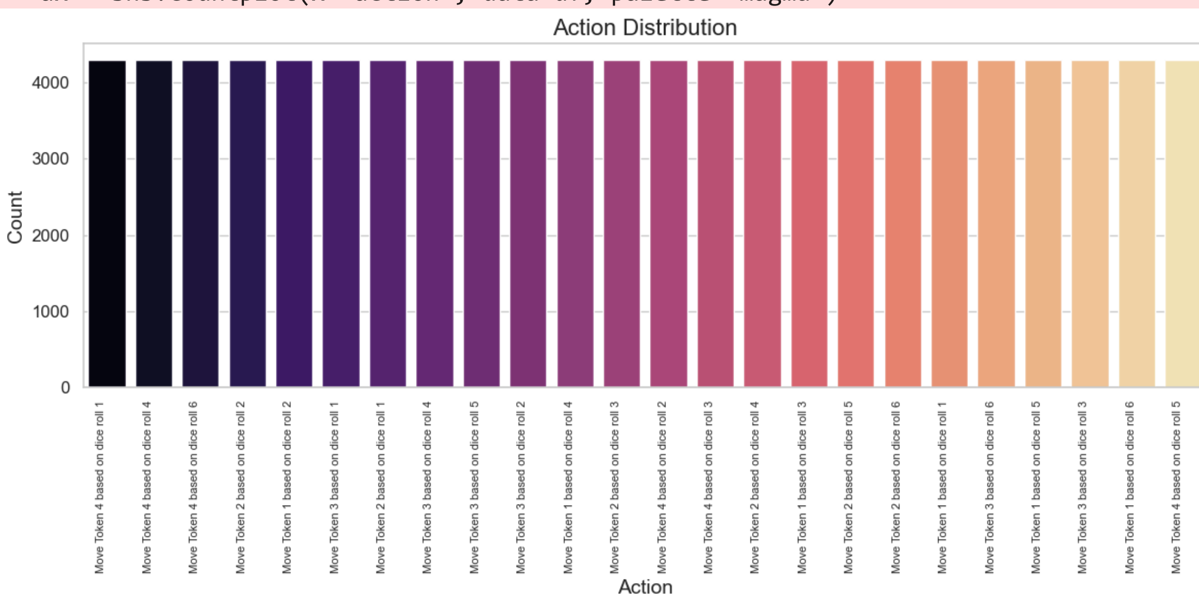
In [48]:

```python
# Cell 39: Data Cleaning and Preprocessing

# First, check which columns actually exist
needed_cols = ['strategy', 'action', 'dice_roll']
existing_cols = [col for col in needed_cols if col in df.columns]

# Only dropna if columns exist
if existing_cols:
    df = df.dropna(subset=existing_cols)

# Encode columns safely
if 'strategy' in df.columns:
    df['strategy_encoded'] = df['strategy'].astype('category').cat.codes

if 'action' in df.columns:
    df['action_encoded'] = df['action'].astype('category').cat.codes

# Scale features if they exist
features = ['dice_roll', 'player_pieces_home', 'player_pieces_finished',
            'opponent_pieces_home', 'opponent_pieces_finished']

features_existing = [f for f in features if f in df.columns]

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()

if features_existing:
    df[features_existing] = scaler.fit_transform(df[features_existing])
```

In [49]:
```python
# Cell 40: Data Visualization (Beautiful Version)

import seaborn as sns
import matplotlib.pyplot as plt

# Set nice style
sns.set(style="whitegrid")

# Plot Strategy Distribution
if 'strategy' in df.columns:
    plt.figure(figsize=(8, 6))
    ax = sns.countplot(x='strategy', data=df, palette='viridis')
    plt.title('Strategy Distribution', fontsize=16)
    plt.xlabel('Strategy', fontsize=14)
    plt.ylabel('Count', fontsize=14)
    plt.xticks(fontsize=12)
    plt.yticks(fontsize=12)
    plt.tight_layout()
    plt.show()
else:
    print("'strategy' column not found.")

# Plot Action Distribution
if 'action' in df.columns:
    plt.figure(figsize=(12, 6))
    ax = sns.countplot(x='action', data=df, palette='magma')
    plt.title('Action Distribution', fontsize=16)
    plt.xlabel('Action', fontsize=14)
    plt.ylabel('Count', fontsize=14)
    plt.xticks(rotation=90, fontsize=8, ha='right')  # Rotate 90 degrees for better
    plt.yticks(fontsize=12)
    plt.tight_layout()
    plt.show()
else:
    print("'action' column not found.")
```

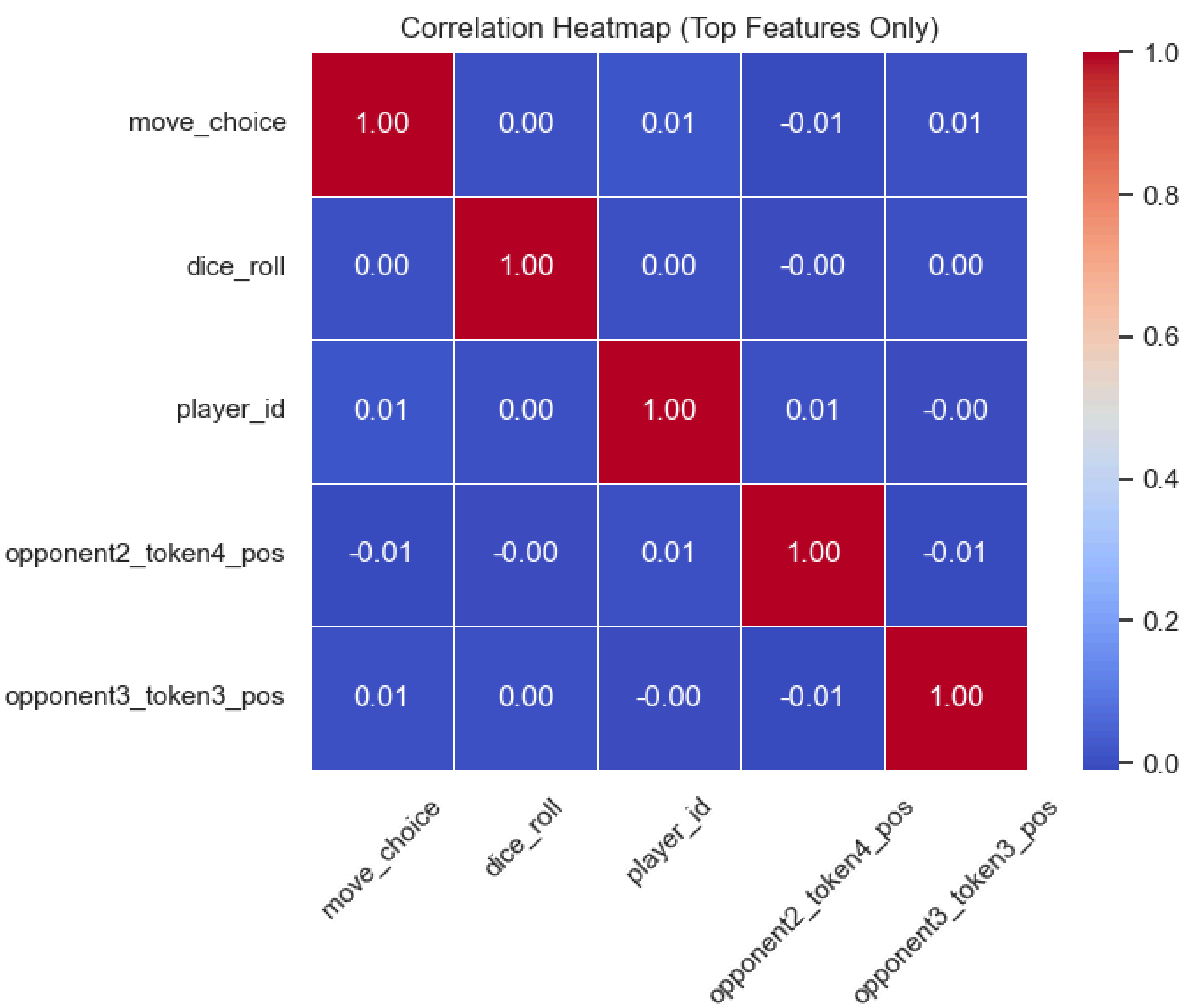
```
C:\Users\Zaid J Adam\AppData\Local\Temp\ipykernel_7576\1523624603.py:12: FutureWarni
ng:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.1
4.0. Assign the `x` variable to `hue` and set `legend=False` for the same effect.

  ax = sns.countplot(x='strategy', data=df, palette='viridis')
```


```
C:\Users\Zaid J Adam\AppData\Local\Temp\ipykernel_7576\1523624603.py:26: FutureWarni
ng:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.1
4.0. Assign the `x` variable to `hue` and set `legend=False` for the same effect.

  ax = sns.countplot(x='action', data=df, palette='magma')
```



In [50]:
```python
# Cell 41: Cleaned Correlation Heatmap (Top Features Only)
# Select only numeric columns
numeric_df = df.select_dtypes(include=[np.number])
```

```
# Compute the correlation matrix
corr_matrix = numeric_df.corr()

# Focus on top correlations with the target (e.g., action_encoded)
target = 'action_encoded'
top_features = corr_matrix[target].abs().sort_values(ascending=False)[1:6].index

# Plot heatmap for selected features
plt.figure(figsize=(8, 6))
sns.heatmap(
    corr_matrix.loc[top_features, top_features],
    annot=True,
    fmt=".2f",
    cmap="coolwarm",
    linewidths=0.5,
    square=True
)
plt.title("Correlation Heatmap (Top Features Only)")
plt.xticks(rotation=45)
plt.yticks(rotation=0)
plt.tight_layout()
plt.show()
```



In [51]:
```python
# Cell 42: Train-Test Split

# Define feature columns
features = ['dice_roll', 'player_pieces_home', 'player_pieces_finished',
            'opponent_pieces_home', 'opponent_pieces_finished', 'strategy_encoded']

# Keep only existing features
features_existing = [col for col in features if col in df.columns]

# Check if 'action_encoded' exists for target
if features_existing and 'action_encoded' in df.columns:
    X = df[features_existing]
    y = df['action_encoded']

    from sklearn.model_selection import train_test_split
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random

    print("Train-Test Split successful!")
else:
    print("Required columns missing for Train-Test Split.")
```

```
Train-Test Split successful!
```

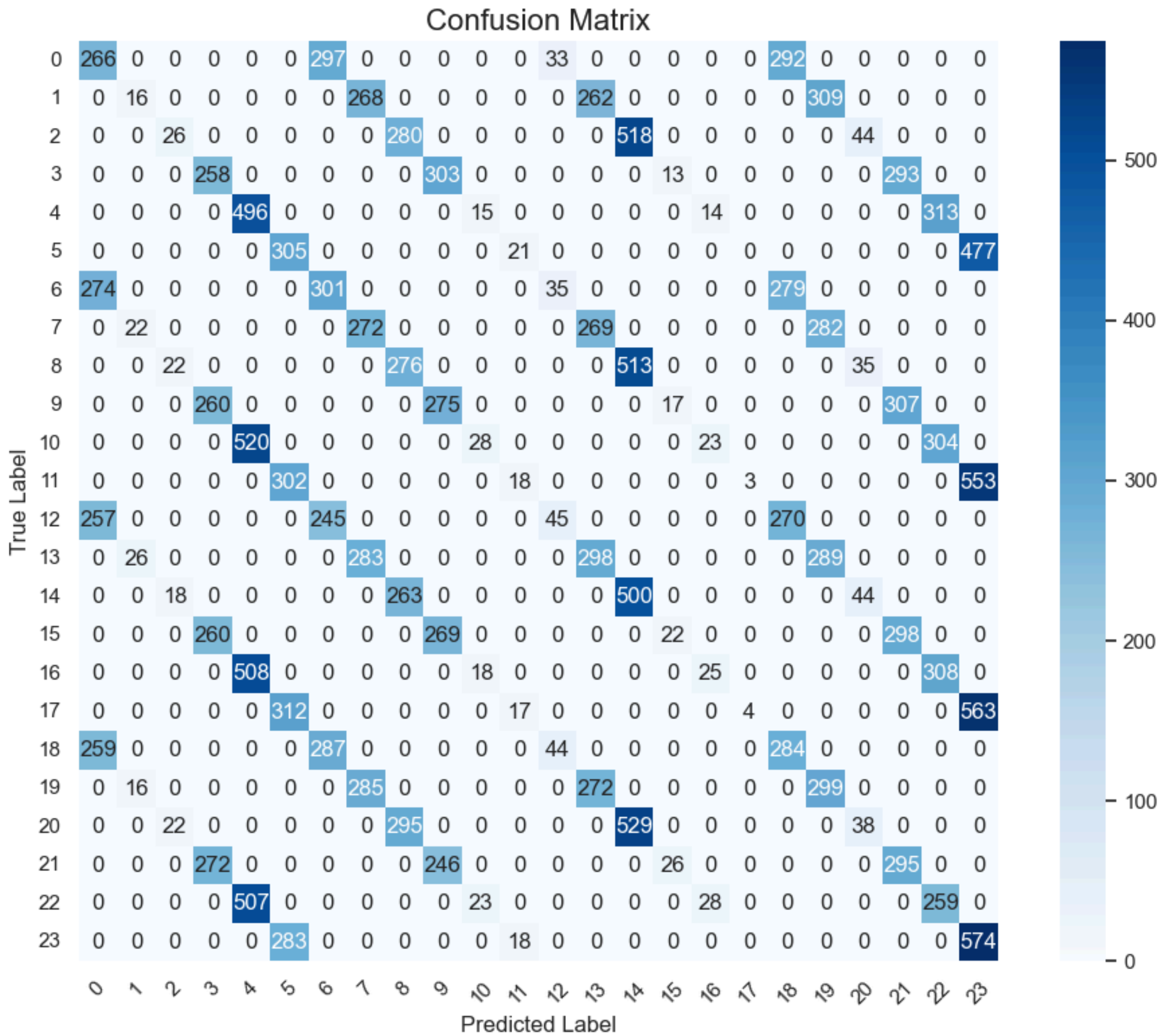
In [52]:
```python
# Cell 43: Train Model
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
```

In [53]:
```python
# Cell 44: Model Evaluation (Beautiful Version)

from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

# Only evaluate if y_test and y_pred are available
if 'y_test' in locals() and 'y_pred' in locals():
    acc = accuracy_score(y_test, y_pred)
    print(f"\nAccuracy: {acc:.4f}\n")

    # Print clean classification report as a DataFrame
    report = classification_report(y_test, y_pred, output_dict=True)
    report_df = pd.DataFrame(report).transpose()
    display(report_df.style.background_gradient(cmap="Blues"))

    # Confusion Matrix
    cm = confusion_matrix(y_test, y_pred)
    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=True, square=True,
                xticklabels=np.unique(y_test), yticklabels=np.unique(y_test))
    plt.title("Confusion Matrix", fontsize=16)
    plt.xlabel("Predicted Label", fontsize=12)
    plt.ylabel("True Label", fontsize=12)
    plt.xticks(rotation=45)
```

```
    plt.yticks(rotation=0)
    plt.tight_layout()
    plt.show()
else:
    print("Cannot evaluate model because y_test or y_pred is missing.")
```

```
Accuracy: 0.2513
```

	precision	recall	f1-score	support
0	0.251894	0.299550	0.273663	888.000000
1	0.200000	0.018713	0.034225	855.000000
2	0.295455	0.029954	0.054393	868.000000
3	0.245714	0.297578	0.269171	867.000000
4	0.244215	0.591885	0.345765	838.000000
5	0.253744	0.379826	0.304239	803.000000
6	0.266372	0.338583	0.298167	889.000000
7	0.245487	0.321893	0.278546	845.000000
8	0.247756	0.326241	0.281633	846.000000
9	0.251601	0.320140	0.281762	859.000000
10	0.333333	0.032000	0.058394	875.000000
11	0.243243	0.020548	0.037895	876.000000
12	0.286624	0.055080	0.092402	817.000000
13	0.270663	0.332589	0.298448	896.000000
14	0.242718	0.606061	0.346620	825.000000
15	0.282051	0.025913	0.047465	849.000000
16	0.277778	0.029104	0.052687	859.000000
17	0.571429	0.004464	0.008859	896.000000
18	0.252444	0.324943	0.284142	874.000000
19	0.253605	0.342890	0.291565	872.000000
20	0.236025	0.042986	0.072727	884.000000
21	0.247276	0.351609	0.290354	839.000000
22	0.218750	0.317013	0.258871	817.000000
23	0.264882	0.656000	0.377383	875.000000
accuracy	0.251310	0.251310	0.251310	0.251310
macro avg	0.270127	0.252732	0.205807	20612.000000
weighted avg	0.270846	0.251310	0.205050	20612.000000



In [54]:

```python
# Cell 45: Reinforcement Learning Setup - Q-Learning Agent
# Define simplified environment for learning token movement strategy

def get_initial_state():
    # Random simplified state: [dice, home, finished, opp_home, opp_finished]
    return tuple(np.random.randint(0, 6, size=5))

def choose_action(state, q_table, epsilon=0.1):
    if random.random() < epsilon or state not in q_table or not q_table[state]:
        return random.choice(range(4))  # 4 actions: one per token
    return max(q_table[state], key=q_table[state].get)



def step(state, action):
    # Simplified outcome of an action
    new_state = tuple(np.random.randint(0, 6, size=5))
    reward = random.choice([1, -1, 5, -2])
    done = random.random() < 0.1  # 10% chance to end game
    return new_state, reward, done

def update_q(q_table, state, action, reward, new_state, alpha=0.1, gamma=0.9):
    if state not in q_table:
        q_table[state] = defaultdict(float)
```

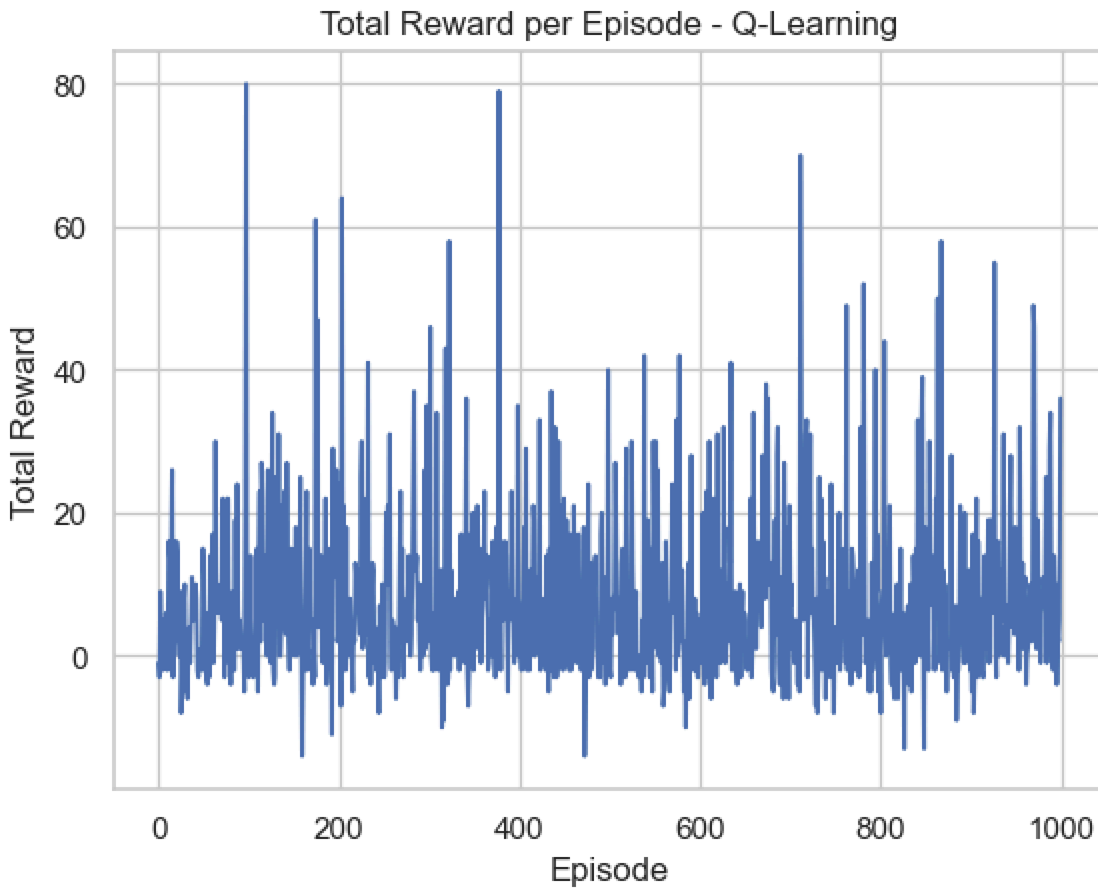
```python
    if new_state not in q_table:
        q_table[new_state] = defaultdict(float)
    best_next = max(q_table[new_state].values(), default=0.0)
    q_table[state][action] += alpha * (reward + gamma * best_next - q_table[state][
```

In [55]:

```python
# Cell 46: Q-Learning Training
q_table = defaultdict(lambda: defaultdict(float))
episodes = 1000
reward_log = []

for episode in range(episodes):
    state = get_initial_state()
    total_reward = 0
    steps = 0
    done = False
    while not done and steps < 100:
        action = choose_action(state, q_table)
        new_state, reward, done = step(state, action)
        update_q(q_table, state, action, reward, new_state)
        state = new_state
        total_reward += reward
        steps += 1
    reward_log.append(total_reward)
```

In [56]:

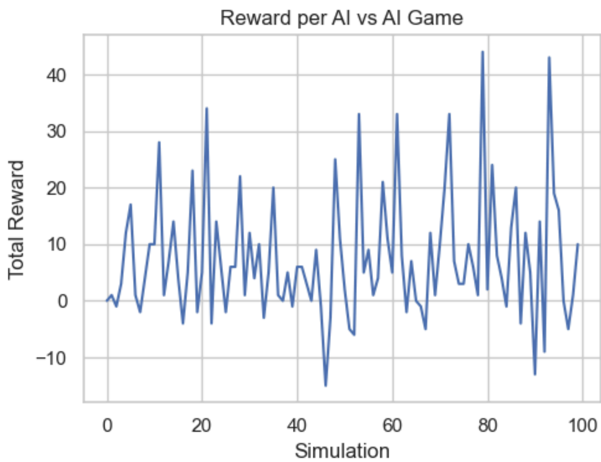
```python
# Cell 47: Plot Learning Progress
plt.plot(reward_log)
plt.title("Total Reward per Episode - Q-Learning")
plt.xlabel("Episode")
plt.ylabel("Total Reward")
plt.grid(True)
plt.show()
```

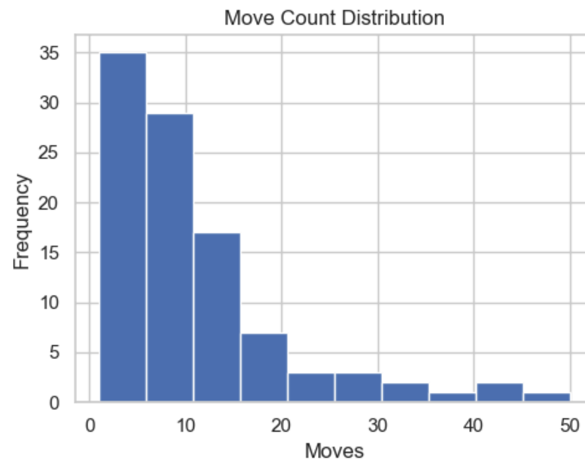


In [57]:
```
# Cell 48: Simulate AI vs AI Using Q-Table

def simulate_ai_vs_ai_game(q_table):
    state = get_initial_state()
    total_reward = 0
    move_count = 0
    done = False
    while not done and move_count < 50:
        action = choose_action(state, q_table, epsilon=0.0)  # greedy policy
        new_state, reward, done = step(state, action)
        total_reward += reward
        state = new_state
        move_count += 1
    return total_reward, move_count

# Run multiple simulations
simulation_results = [simulate_ai_vs_ai_game(q_table) for _ in range(100)]
sim_rewards, sim_moves = zip(*simulation_results)
```

In [58]:
```
# Cell 49: Performance Analysis - AI vs AI
plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1)
plt.plot(sim_rewards)
plt.title("Reward per AI vs AI Game")
plt.xlabel("Simulation")
plt.ylabel("Total Reward")

plt.subplot(1, 2, 2)
```

```
plt.hist(sim_moves)
plt.title("Move Count Distribution")
plt.xlabel("Moves")
plt.ylabel("Frequency")
plt.tight_layout()
plt.show()

print("Average Total Reward:", np.mean(sim_rewards))
print("Average Moves per Game:", np.mean(sim_moves))
```






```
Average Total Reward: 7.42
Average Moves per Game: 10.62
```