ai-chess-agent

December 5, 2019

[1]:
```python
import time
import chess
from IPython.display import display, HTML, clear_output
import numpy as np
import pandas as pd
import timeit
import random
```

1 AI Chess Agent Project

1.1 helper functions

1.1.1 Displays the chess board

[2]:
```python
def display_board(board, use_svg):
    if use_svg:
        return board._repr_svg_()
    else:
        return "<pre>" + str(board) + "</pre>"
```

1.1.2 Checks if player agent is white or black

[3]:
```python
def who(agent):
    return "White" if agent == chess.WHITE else "Black"
```

1.1.3 Obtains available moves

[4]:
```python
def get_move(prompt):
    uci = input(prompt)
    if uci and uci[0] == "q":
        raise KeyboardInterrupt()
    try:
        chess.Move.from_uci(uci)
    except:
        uci = None
    return uci
```

1.1.4 Tallies the white and black players pieces

[5]:
```
def count_pieces(board):
    num_pieces = [0,0]

    num_pieces[0] += len(board.pieces(chess.PAWN,   chess.WHITE))
    num_pieces[0] += len(board.pieces(chess.BISHOP, chess.WHITE))
    num_pieces[0] += len(board.pieces(chess.KING,   chess.WHITE))
    num_pieces[0] += len(board.pieces(chess.QUEEN,  chess.WHITE))
    num_pieces[0] += len(board.pieces(chess.KNIGHT, chess.WHITE))
    num_pieces[0] += len(board.pieces(chess.ROOK,   chess.WHITE))

    num_pieces[1] += len(board.pieces(chess.PAWN,   chess.BLACK))
    num_pieces[1] += len(board.pieces(chess.BISHOP, chess.BLACK))
    num_pieces[1] += len(board.pieces(chess.KING,   chess.BLACK))
    num_pieces[1] += len(board.pieces(chess.QUEEN,  chess.BLACK))
    num_pieces[1] += len(board.pieces(chess.KNIGHT, chess.BLACK))
    num_pieces[1] += len(board.pieces(chess.ROOK,   chess.BLACK))

    return num_pieces
```

1.1.5 Plays a single game with two agent players

[6]:
```
def play_game(agent1, agent2, visual="svg", pause=0.1):
    """
    agentN1, agent2: functions that takes board, return uci move
    visual: "simple" | "svg" | None
    """
    use_svg = (visual == "svg")
    board = chess.Board()
    try:
        while not board.is_game_over(claim_draw=True):
            if board.turn == chess.WHITE:
                uci = agent1(board)
            else:
                uci = agent2(board)
            name = who(board.turn)
            board.push_uci(uci)
            board_stop = display_board(board, use_svg)
            html = "<b>Move %s %s, Play '%s':</b><br/>%s" % (
                       len(board.move_stack), name, uci, board_stop)
            if visual is not None:
                if visual == "svg":
                    clear_output(wait=True)
                display(HTML(html))
                if visual == "svg":
                    time.sleep(pause)
    except KeyboardInterrupt:
        msg = "Game interrupted!"
        return (False, msg, board)
    game_has_winner = False
    if board.is_checkmate():
        msg = "checkmate: " + who(not board.turn) + " wins!"
        game_has_winner = not board.turn
    elif board.is_stalemate():
        msg = "draw: stalemate"
    elif board.is_fivefold_repetition():
        msg = "draw: 5-fold repetition"
    elif board.is_insufficient_material():
        msg = "draw: insufficient material"
    elif board.can_claim_draw():
        msg = "draw: claim"
    if visual is not None:
        print(msg)

    return (game_has_winner, msg, board)
```

1.1.6 "Driver" allows for two agent players to play multiple games for a provided number of iterations. Returns a list of scores

[7]:

```
def run(agent1, agent2, iterations, agent1_name, agent2_name):
#     df_scoreboard = pd.DataFrame(data={},␣
 ↪columns=['game_result','winner','moves_played',␣
 ↪'remaining_w_pieces','remaining_b_pieces'])
    scores_list = list()

    for round_num in range(iterations):

        terminal_state = play_game(agent1, agent2, visual="svg", pause=0.001)
#         time = timeit.timeit(play_game(agent1, agent2, visual="svg",␣
 ↪pause=0), number=100)/100

        game_hase_winner = terminal_state[0]
        msg = terminal_state[1]
        moves_played = len(terminal_state[2].move_stack)
        remaining_w_pieces = count_pieces(terminal_state[2])[0]
        remaining_b_pieces = count_pieces(terminal_state[2])[1]

#         result_list = (game_hase_winner, msg, moves_played,␣
 ↪count_pieces(result[2])[0], count_pieces(result[2])[1], result[3])
        result_list = (round_num + 1, iterations, agent1_name, agent2_name,␣
 ↪game_hase_winner, msg, moves_played, remaining_w_pieces, remaining_b_pieces)
        scores_list.append(result_list)
    return scores_list
```

1.1.7 Results Scoreboard

```
[8]: df = pd.DataFrame(columns=['round_num', 'iterations', 'agent1_name',␣
      ↪'agent2_name','game_has_winner','winner','moves_played',␣
      ↪'remain_w_pieces','remaining_b_pieces'])
```

1.2 Random Agent Evaluation

1.2.1 plays two random agents against eachother 10 times

1.2.2 Random Agent player

```
[9]: def random_agent(board):
         move = random.choice(list(board.legal_moves))
         return move.uci()
```

```
[10]: rand_eval_scores = run(random_agent, random_agent, 10, "random_agent",␣
       ↪"random_agent")
```

```
<IPython.core.display.HTML object>


draw: claim
```

```
[11]: df_rand_eval_scoreboard = pd.DataFrame(data=rand_eval_scores,␣
       ↪columns=['round_num', 'iterations', 'agent1_name',␣
       ↪'agent2_name','game_has_winner','winner','moves_played',␣
       ↪'remain_w_pieces','remaining_b_pieces'])

      df_rand_eval_scoreboard.sort_values(by=['moves_played'], inplace=False,␣
       ↪ascending=True)
```

```
[11]:    round_num  iterations   agent1_name   agent2_name  game_has_winner  \
      1          2          10  random_agent  random_agent            False
      8          9          10  random_agent  random_agent            False
      4          5          10  random_agent  random_agent            False
      5          6          10  random_agent  random_agent            False
      6          7          10  random_agent  random_agent            False
      0          1          10  random_agent  random_agent            False
      2          3          10  random_agent  random_agent            False
      9         10          10  random_agent  random_agent            False
      7          8          10  random_agent  random_agent            False
      3          4          10  random_agent  random_agent            False

                           winner  moves_played  remain_w_pieces  \
      1       checkmate: Black wins!            80               10
```

```
8       checkmate: Black wins!           130                7
4  draw: insufficient material           273                2
5                  draw: claim           311                5
6                  draw: claim           333                1
0                  draw: claim           365                2
2  draw: insufficient material           396                2
9                  draw: claim           420                2
7                  draw: claim           421                1
3                  draw: claim           448                2

   remaining_b_pieces
1                  12
8                   6
4                   1
5                   1
6                   3
0                   2
2                   1
9                   2
7                   5
3                   1
```

[12]:

```
#update results scoreboard
df = df.append(df_rand_eval_scoreboard, ignore_index=True)
```

1.2.3 Scoreboard

[13]:

```
#10 best games by moves_played ascending
df.sort_values(by=['moves_played'], inplace=False, ascending=True).head(10)
```

[13]:

```
  round_num iterations   agent1_name   agent2_name game_has_winner  \
1         2         10  random_agent  random_agent           False
8         9         10  random_agent  random_agent           False
4         5         10  random_agent  random_agent           False
5         6         10  random_agent  random_agent           False
6         7         10  random_agent  random_agent           False
0         1         10  random_agent  random_agent           False
2         3         10  random_agent  random_agent           False
9        10         10  random_agent  random_agent           False
7         8         10  random_agent  random_agent           False
3         4         10  random_agent  random_agent           False

                        winner moves_played remain_w_pieces remaining_b_pieces
1       checkmate: Black wins!           80              10                 12
8       checkmate: Black wins!          130               7                  6
4  draw: insufficient material          273               2                  1
5                  draw: claim          311               5                  1
```

```
6                  draw: claim          333               1                  3
0                  draw: claim          365               2                  2
2  draw: insufficient material          396               2                  1
9                  draw: claim          420               2                  2
7                  draw: claim          421               1                  5
3                  draw: claim          448               2                  1
```

[14]:
```
# #update results scoreboard
# df.append(df_rand_eval_scoreboard, ignore_index=True)
```

1.3 Naive Agent Evaluation

1.3.1 Naive evaluation function

Sets the score to 0 and assigns weights to every piece on the board. The weighted sum of all the available pieces on the board is then computed.

The white pieces are assigned positive values while the black ones are assigned negative values of the same magnitude.

[15]:
```
def naive_eval(board, move, my_color):
    score = 0
    ## Check some things about this move:
    # score += 10 if board.is_capture(move) else 0
    # To actually make the move:
    board.push(move)
    # Now check some other things:
    for (piece, value) in [(chess.PAWN, 1),
                           (chess.BISHOP, 4),
                           (chess.KING, 0),
                           (chess.QUEEN, 10),
                           (chess.KNIGHT, 5),
                           (chess.ROOK, 3)]:
        score += len(board.pieces(piece, my_color)) * value
        score -= len(board.pieces(piece, not my_color)) * value
        # can also check things about the pieces position here
    return score
```

1.3.2 Naive Agent

Chooses best score

[16]:
```
def naive_agent(board):
    moves = list(board.legal_moves)
    for move in moves:
        newboard = board.copy()
        # go through board and return a score
        move.score = naive_eval(newboard, move, board.turn)
    moves.sort(key=lambda move: move.score, reverse=True) # sort on score
    return moves[0].uci()
```

[17]:
```
# result = play_game(random_agent, naive_agent, visual="svg", pause=0)
```

[18]:
```
naive_eval_scores = run(naive_agent, random_agent, 10, "naive_agent",␣
 ↪"random_agent")
```

```
<IPython.core.display.HTML object>


draw: claim
```

[19]:
```
# df_naive_eval_scoreboard = pd.DataFrame(data=naive_eval_scores,␣
 ↪columns=['round_num', 'iterations', 'agent1_name',␣
 ↪'agent2_name','game_has_winner','winner','moves_played',␣
 ↪'remain_w_pieces','remaining_b_pieces'])
# df_naive_eval_scoreboard

df_naive_eval_scoreboard = pd.DataFrame(data=naive_eval_scores,␣
 ↪columns=['round_num', 'iterations', 'agent1_name',␣
 ↪'agent2_name','game_has_winner','winner','moves_played',␣
 ↪'remain_w_pieces','remaining_b_pieces'])

df_naive_eval_scoreboard.sort_values(by=['moves_played'], inplace=False,␣
 ↪ascending=True)
```

[19]:
```
   round_num  iterations  agent1_name   agent2_name  game_has_winner  \
9         10          10  naive_agent  random_agent            False
6          7          10  naive_agent  random_agent            False
1          2          10  naive_agent  random_agent            False
8          9          10  naive_agent  random_agent            False
4          5          10  naive_agent  random_agent            False
5          6          10  naive_agent  random_agent            False
2          3          10  naive_agent  random_agent            False
7          8          10  naive_agent  random_agent            False
0          1          10  naive_agent  random_agent            False
3          4          10  naive_agent  random_agent            False

        winner  moves_played  remain_w_pieces  remaining_b_pieces
9  draw: claim            35               16                   6
6  draw: claim            39               15                  10
1  draw: claim            41               15                   3
8  draw: claim            41               15                  12
4  draw: claim            77               13                   2
5  draw: claim            81               16                   1
2  draw: claim            87               15                   3
```

```
7  draw: claim            95               15                   1
0  draw: claim            97               13                   1
3  draw: claim           107               10                   1
```

[20]:

```
# #update results scoreboard
# df.append(df_naive_eval_scoreboard, ignore_index=True)

#update results scoreboard
df = df.append(df_naive_eval_scoreboard, ignore_index=True)
```

1.3.3 Scoreboard: Top 10 Games With Fewest Moves

[21]:

```
#10 best games by moves_played ascending
df.sort_values(by=['moves_played'], inplace=False, ascending=True).head(10)
```

[21]:

```
   round_num iterations   agent1_name   agent2_name game_has_winner  \
19        10         10   naive_agent  random_agent           False
16         7         10   naive_agent  random_agent           False
18         9         10   naive_agent  random_agent           False
11         2         10   naive_agent  random_agent           False
14         5         10   naive_agent  random_agent           False
1          2         10  random_agent  random_agent           False
15         6         10   naive_agent  random_agent           False
12         3         10   naive_agent  random_agent           False
17         8         10   naive_agent  random_agent           False
10         1         10   naive_agent  random_agent           False

                    winner moves_played remain_w_pieces remaining_b_pieces
19             draw: claim           35              16                  6
16             draw: claim           39              15                 10
18             draw: claim           41              15                 12
11             draw: claim           41              15                  3
14             draw: claim           77              13                  2
1   checkmate: Black wins!           80              10                 12
15             draw: claim           81              16                  1
12             draw: claim           87              15                  3
17             draw: claim           95              15                  1
10             draw: claim           97              13                  1
```

1.3.4 Naive Agent With Improved Evaluation

1.4 Naive Random Heuristic Evaluation

Sets the score to a random value and assigns weights to every piece on the board. The weighted sum of all the available pieces on the board is then computed.

The white pieces are assigned positive values while the black ones are assigned negative values of the same magnitude.

[22]:
```
def naive_random_heuristic_eval(board, move, my_color):
    score = random.random()
    ## Check some things about this move:
    # score += 10 if board.is_capture(move) else 0
    # To actually make the move:
    board.push(move)
    # Now check some other things:
    for (piece, value) in [(chess.PAWN, 1),
                           (chess.BISHOP, 4),
                           (chess.KING, 0),
                           (chess.QUEEN, 10),
                           (chess.KNIGHT, 5),
                           (chess.ROOK, 3)]:
        score += len(board.pieces(piece, my_color)) * value
        score -= len(board.pieces(piece, not my_color)) * value
        # can also check things about the pieces position here
    # Check global things about the board
    score += 100 if board.is_checkmate() else 0
    return score
```

1.4.1 Naive Agent with Random Heuristic Evaluator

Chooses best score

[23]:
```
def naive_random_heuristic_agent(board):
    moves = list(board.legal_moves)
    for move in moves:
        newboard = board.copy()
        # go through board and return a score
        move.score = naive_random_heuristic_eval(newboard, move, board.turn)
    moves.sort(key=lambda move: move.score, reverse=True) # sort on score
    return moves[0].uci()
```

[24]:
```
naive_rand_heuristic_eval_scores = run(naive_random_heuristic_agent,␣
 ↪random_agent, 10, "naive_random_heuristic_agent", "random_agent")
```

```
<IPython.core.display.HTML object>


checkmate: White wins!
```

[25]:
```
# df_naive_rand_heuristic_eval_scoreboard = pd.
 ↪DataFrame(data=naive_rand_heuristic_eval_scores, columns=['round_num',␣
 ↪'iterations', 'agent1_name',␣
 ↪'agent2_name','game_has_winner','winner','moves_played',␣
 ↪'remain_w_pieces','remaining_b_pieces'])
# df_naive_rand_heuristic_eval_scoreboard
```

```
df_naive_rand_heuristic_eval_scoreboard = pd.
 ↪DataFrame(data=naive_rand_heuristic_eval_scores, columns=['round_num',␣
 ↪'iterations', 'agent1_name',␣
 ↪'agent2_name','game_has_winner','winner','moves_played',␣
 ↪'remain_w_pieces','remaining_b_pieces'])
df_naive_rand_heuristic_eval_scoreboard.sort_values(by=['moves_played'],␣
 ↪inplace=False, ascending=True)
```

[25]:

```
   round_num  iterations                   agent1_name   agent2_name  \
2          3          10  naive_random_heuristic_agent  random_agent
6          7          10  naive_random_heuristic_agent  random_agent
3          4          10  naive_random_heuristic_agent  random_agent
9         10          10  naive_random_heuristic_agent  random_agent
7          8          10  naive_random_heuristic_agent  random_agent
5          6          10  naive_random_heuristic_agent  random_agent
8          9          10  naive_random_heuristic_agent  random_agent
4          5          10  naive_random_heuristic_agent  random_agent
1          2          10  naive_random_heuristic_agent  random_agent
0          1          10  naive_random_heuristic_agent  random_agent

   game_has_winner                  winner  moves_played  remain_w_pieces  \
2             True  checkmate: White wins!            65               11
6             True  checkmate: White wins!            65               15
3             True  checkmate: White wins!            67               14
9             True  checkmate: White wins!            69               14
7             True  checkmate: White wins!           105                9
5            False         draw: stalemate           107               13
8             True  checkmate: White wins!           133               10
4             True  checkmate: White wins!           155               12
1            False             draw: claim           214               10
0             True  checkmate: White wins!           221                6

   remaining_b_pieces
2                   2
6                   3
3                   1
9                   2
7                   1
5                   1
8                   1
4                   1
1                   1
0                   1
```

[26]:

```
#update results scoreboard
df = df.append(df_naive_rand_heuristic_eval_scoreboard , ignore_index=True)
```

1.4.2 Scoreboard: Top 10 Games With Fewest Moves

[27]:

```
#10 best games by moves_played ascending
df.sort_values(by=['moves_played'], inplace=False, ascending=True).head(10)
```

[27]:

```
   round_num iterations                   agent1_name   agent2_name  \
19        10         10                   naive_agent  random_agent
16         7         10                   naive_agent  random_agent
18         9         10                   naive_agent  random_agent
11         2         10                   naive_agent  random_agent
26         7         10  naive_random_heuristic_agent  random_agent
22         3         10  naive_random_heuristic_agent  random_agent
23         4         10  naive_random_heuristic_agent  random_agent
29        10         10  naive_random_heuristic_agent  random_agent
14         5         10                   naive_agent  random_agent
1          2         10                  random_agent  random_agent

   game_has_winner                    winner moves_played remain_w_pieces  \
19           False               draw: claim           35              16
16           False               draw: claim           39              15
18           False               draw: claim           41              15
11           False               draw: claim           41              15
26            True    checkmate: White wins!           65              15
22            True    checkmate: White wins!           65              11
23            True    checkmate: White wins!           67              14
29            True    checkmate: White wins!           69              14
14           False               draw: claim           77              13
1            False    checkmate: Black wins!           80              10

   remaining_b_pieces
19                  6
16                 10
18                 12
11                  3
26                  3
22                  2
23                  1
29                  2
14                  2
1                  12
```

1.5 Minimax

1.6 minimax evaluation

Sets the score to a random value and assigns weights to every piece on the board. The weighted sum of all the available pieces on the board is then computed.

The white pieces are assigned positive values while the black ones are assigned negative values of the same magnitude.

[28]:

```
def minimax_eval(board):
    # moves = list(board.legal_moves)
    # for move in moves:
    #     newboard = board.copy()
    #     # go through board and return a score
    #     move.score = staticAnalysis(newboard, move, board.turn)
    # moves.sort(key=lambda move: move.score, reverse=True) # sort on score
    # return moves[0].uci()
    score = random.random()
    for (piece, value) in [(chess.PAWN, 1),
                          (chess.BISHOP, 4),
                          (chess.KING, 0),
                          (chess.QUEEN, 10),
                          (chess.KNIGHT, 5),
                          (chess.ROOK, 3)]:
        score += len(board.pieces(piece, True)) * value
        score -= len(board.pieces(piece,False)) * value
        # can also check things about the pieces position here
    return score
```

[29]:

```
def maxValue(board, currentAgent, depth):
    bestMove = -9999

    moves = list(board.legal_moves)
    for move in moves:
        newboard = board.copy()
        newboard.push_uci(move.uci())
        result = miniMaxDecision(newboard, not currentAgent , depth -1)
        if result > bestMove:
            bestMove = result
    return bestMove
```

[30]:

```
def minValue(board, currentAgent, depth):
    bestMove = 9999

    moves = list(board.legal_moves)
    for move in moves:
        newboard = board.copy()
        newboard.push_uci(move.uci())
        result = miniMaxDecision(newboard, not currentAgent, depth -1)
        if result < bestMove:
            bestMove = result
    return bestMove
```

```
[31]: def miniMaxDecision(board, currentAgent, depth):
          if depth == 0 :
              return minimax_eval(board)

          if currentAgent:
              return maxValue(board, not currentAgent, depth - 1)
          else:
              return minValue(board, not currentAgent, depth - 1)
```

```
[32]: def mini_max_agent(board):
          moves = list(board.legal_moves)
          for move in moves:
              newboard = board.copy()
              newboard.push_uci(move.uci())
              move.score = miniMaxDecision(newboard, False , 2)
          moves.sort(key=lambda move: move.score, reverse=True) # sort on score
          return moves[0].uci()
```

```
[33]: minimax_eval_scores = run(mini_max_agent, random_agent, 10, "mini_max_agent",␣
       ↪"random_agent")
```

```
<IPython.core.display.HTML object>


draw: stalemate
```

```
[34]: df_minimax_eval_scoreboard = pd.DataFrame(data=minimax_eval_scores,␣
       ↪columns=['round_num', 'iterations', 'agent1_name',␣
       ↪'agent2_name','game_has_winner','winner','moves_played',␣
       ↪'remain_w_pieces','remaining_b_pieces'])

      df_minimax_eval_scoreboard.sort_values(by=['moves_played'], inplace=False,␣
       ↪ascending=True)
```

```
[34]:    round_num  iterations     agent1_name   agent2_name  game_has_winner  \
      7          8          10  mini_max_agent  random_agent             True
      5          6          10  mini_max_agent  random_agent             True
      0          1          10  mini_max_agent  random_agent             True
      1          2          10  mini_max_agent  random_agent             True
      2          3          10  mini_max_agent  random_agent             True
      9         10          10  mini_max_agent  random_agent            False
      3          4          10  mini_max_agent  random_agent             True
      6          7          10  mini_max_agent  random_agent             True
      8          9          10  mini_max_agent  random_agent            False
      4          5          10  mini_max_agent  random_agent            False

                         winner  moves_played  remain_w_pieces  remaining_b_pieces
      7  checkmate: White wins!            17               16                  10
```

```
5   checkmate: White wins!           37                16                    8
0   checkmate: White wins!           47                15                    3
1   checkmate: White wins!           49                16                    4
2   checkmate: White wins!           57                14                    6
9          draw: stalemate           57                15                    2
3   checkmate: White wins!           63                16                    2
6   checkmate: White wins!           69                15                    4
8          draw: stalemate           77                16                    3
4          draw: stalemate           93                14                    1
```

[35]:

```
#update results scoreboard
df = df.append(df_minimax_eval_scoreboard, ignore_index=True)
```

1.6.1 Scoreboard: Top 10 Games With Fewest Moves

[36]:

```
#10 best games by moves_played ascending
df.sort_values(by=['moves_played'], inplace=False, ascending=True).head(10)
```

[36]:

```
    round_num iterations    agent1_name   agent2_name game_has_winner  \
37          8         10  mini_max_agent  random_agent            True
19         10         10     naive_agent  random_agent           False
35          6         10  mini_max_agent  random_agent            True
16          7         10     naive_agent  random_agent           False
18          9         10     naive_agent  random_agent           False
11          2         10     naive_agent  random_agent           False
30          1         10  mini_max_agent  random_agent            True
31          2         10  mini_max_agent  random_agent            True
32          3         10  mini_max_agent  random_agent            True
39         10         10  mini_max_agent  random_agent           False

                    winner moves_played remain_w_pieces remaining_b_pieces
37  checkmate: White wins!           17              16                 10
19             draw: claim           35              16                  6
35  checkmate: White wins!           37              16                  8
16             draw: claim           39              15                 10
18             draw: claim           41              15                 12
11             draw: claim           41              15                  3
30  checkmate: White wins!           47              15                  3
31  checkmate: White wins!           49              16                  4
32  checkmate: White wins!           57              14                  6
39         draw: stalemate           57              15                  2
```

1.6.2 Games Where Player 1 (white) Wins, Ordered by Moves Played Desc

[37]:

```
d2 = df.loc[df['winner'] == 'checkmate: White wins!']

# df.sort_values(by=['winner','moves_played'], inplace=False, ascending=True).
↪head(10)
```

[38]:

```
d2.sort_values(by=['moves_played'], inplace=False, ascending=True)
```

[38]:

```
   round_num iterations                  agent1_name   agent2_name  \
37         8         10               mini_max_agent  random_agent   
35         6         10               mini_max_agent  random_agent   
30         1         10               mini_max_agent  random_agent   
31         2         10               mini_max_agent  random_agent   
32         3         10               mini_max_agent  random_agent   
33         4         10               mini_max_agent  random_agent   
22         3         10 naive_random_heuristic_agent  random_agent   
26         7         10 naive_random_heuristic_agent  random_agent   
23         4         10 naive_random_heuristic_agent  random_agent   
29        10         10 naive_random_heuristic_agent  random_agent   
36         7         10               mini_max_agent  random_agent   
27         8         10 naive_random_heuristic_agent  random_agent   
28         9         10 naive_random_heuristic_agent  random_agent   
24         5         10 naive_random_heuristic_agent  random_agent   
20         1         10 naive_random_heuristic_agent  random_agent   

   game_has_winner                  winner moves_played remain_w_pieces  \
37            True  checkmate: White wins!           17              16   
35            True  checkmate: White wins!           37              16   
30            True  checkmate: White wins!           47              15   
31            True  checkmate: White wins!           49              16   
32            True  checkmate: White wins!           57              14   
33            True  checkmate: White wins!           63              16   
22            True  checkmate: White wins!           65              11   
26            True  checkmate: White wins!           65              15   
23            True  checkmate: White wins!           67              14   
29            True  checkmate: White wins!           69              14   
36            True  checkmate: White wins!           69              15   
27            True  checkmate: White wins!          105               9   
28            True  checkmate: White wins!          133              10   
24            True  checkmate: White wins!          155              12   
20            True  checkmate: White wins!          221               6   

   remaining_b_pieces  
37                 10  
35                  8  
30                  3  
```

```
31                    4
32                    6
33                    2
22                    2
26                    3
23                    1
29                    2
36                    4
27                    1
28                    1
24                    1
20                    1
```

```
[1]: import os
     print(os.environ['PATH'])
```

```
/anaconda3/bin:/anaconda3/condabin:/usr/bin:/bin:/usr/sbin:/sbin
```

```
[4]: !export PATH=/Library/TeX/texbin/xelatex:$PATH
```

```
[ ]:
```