Issue - after 1 full game of chess, further chess games ran with genetic algorithm implementation result in an immediate win or loss for one of the sides after only a single move.

Chess implementation explained:

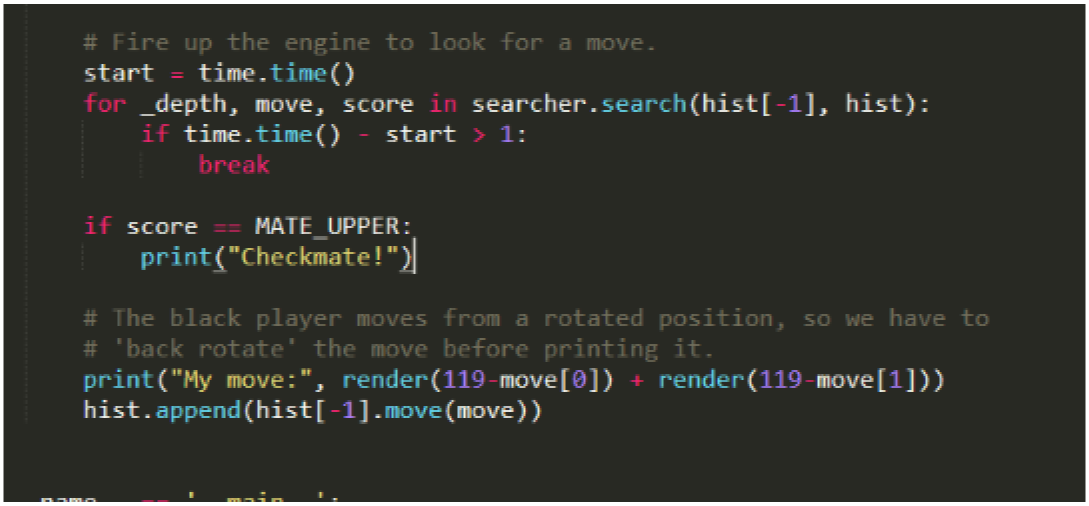
Sunfish:

Classes:

Sunfish has 2 classes built in, a Position class, and a Searcher class. A searcher will search for the best moves, while a position will be scored, have legal moves, etc. Sunfish also had a main method with code seen in figure 1 (see end of document)

Main method:

The main method for sunfish has the following code to account for engine movement:



```
    # Fire up the engine to look for a move.
    start = time.time()
    for _depth, move, score in searcher.search(hist[-1], hist):
        if time.time() - start > 1:
            break

    if score == MATE_UPPER:
        print("Checkmate!")

    # The black player moves from a rotated position, so we have to
    # 'back rotate' the move before printing it.
    print("My move:", render(119-move[0]) + render(119-move[1]))
    hist.append(hist[-1].move(move))


__name__ == '__main__':
```

The searcher first searches for the best position it can find (supposedly) and then adds this move to the ‘hist’ denoting history of positions, where each index contains a position in the game. Note that the comment states that the computer moves from a “Rotated” position. In order to rotate this position back, there exists a .rotate() method which is called within the .move() method evoked on the last line on the position class.

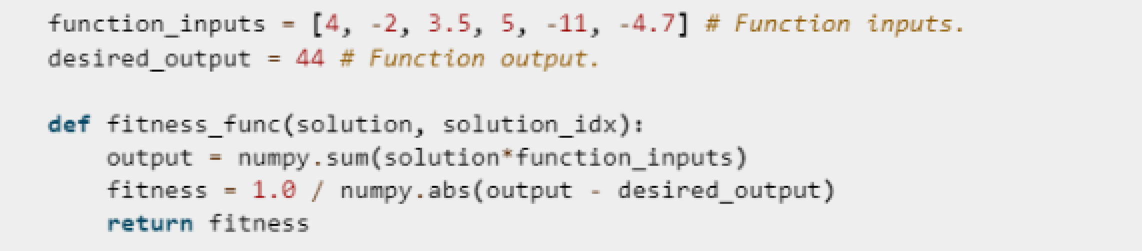
Additional info:

Sunfish uses a GLOBAL pst value which is padded from its original dictionary form in order to be used in the engine

My attempted adaptation:

Pygad:

Pygad uses a fitness function which should take in a solution and a solution_idx, and return a value representing its fitness. Use of the solution_idx variable does not seem required per documentation:



```python
function_inputs = [4, -2, 3.5, 5, -11, -4.7] # Function inputs.
desired_output = 44 # Function output.

def fitness_func(solution, solution_idx):
    output = numpy.sum(solution*function_inputs)
    fitness = 1.0 / numpy.abs(output - desired_output)
    return fitness
```

The inputs are required to come in as a 1d array.

My fitness function (figure 2):

This fitness function essentially converts a solution to a piece square table, taking the 1d array and turning it into a 6x64 dict of piece square tables. A score variable is then set to 0, and incremented if the solution achieves a desirable outcome. The solution is coded to play 3 games using a ‘select_fish’ function (potential location of the issue) which should return a result of the game, a position, and whose turn it was.

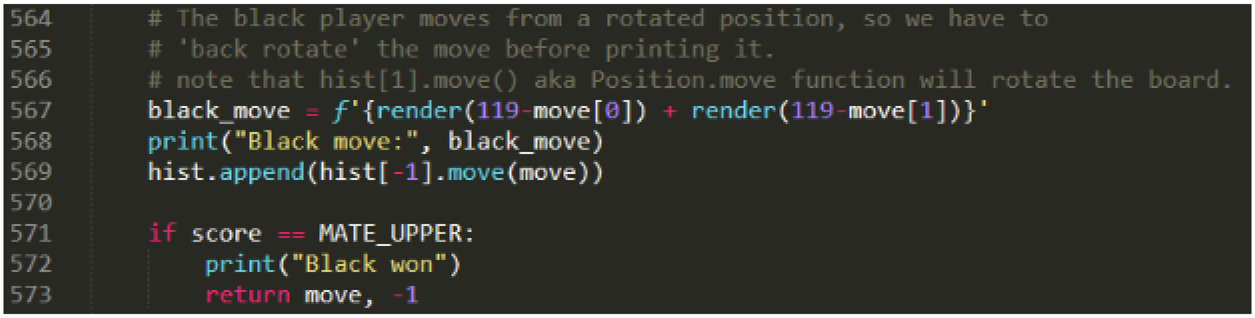
Select_fish function (figure 3):

This function runs 3 times in the fitness function. Just like the main sunfish.py function, it initializes a new history with a default starting position, and a new searcher (searchers seem to contain history and such, and a new one is initialized during the sunfish main function, so i thought it necessary). A null black move and gamestate pair is generated as well for the functions move_white and move_black (also potential failure points?) While the counter is less than our move limit we have defined (80) and the game is not over (stalemate, checkmate) set the GLOBAL value of the piece square table to the appropriate table passed in, and make a move. Remember, sunfish uses a GLOBAL pst variable, so I am passing in specific pst’s into the function and overwriting the global pst with them depending on whose turn it is. This function will return the game result (0, 1, -1, 10, -10 where +-1 denotes checkmate, +-10 denotes stalemate/draw, and 0 denotes no conclusion resorting to stockfish evaluation)

Movement functions (figure 4 - a white, b black):

Each function takes in a history of moves (which is referred to and added to by the modified sunfish program), a searcher (initialized within the select fish function, so the same engine is searching with its history but using different global PST values for the moves it makes), a move made by the previous color (checks for mate again for them before making a move), and a gamestate. It will return a move and a gamestate.

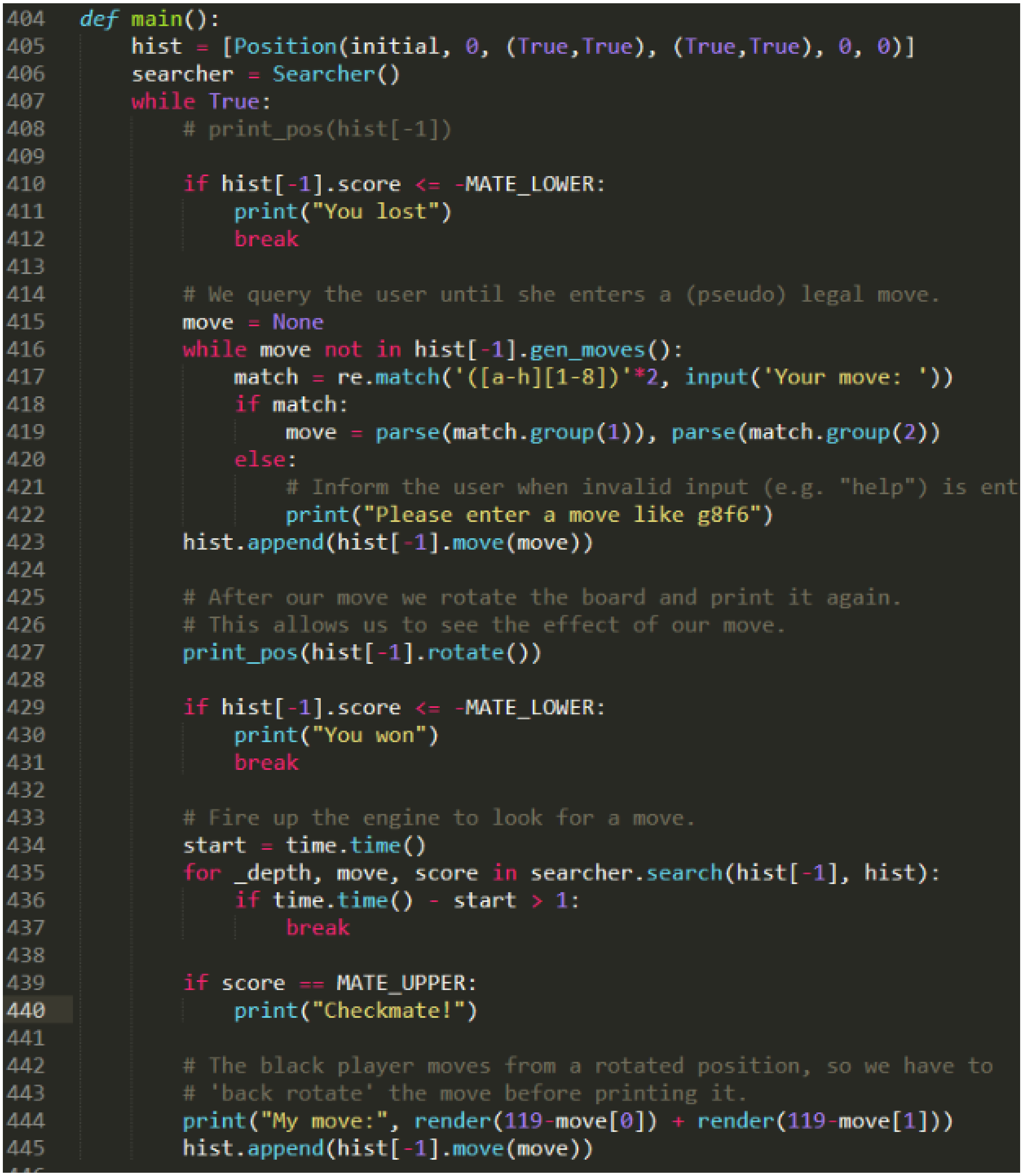
These are essentially the main function of sunfish.py reworked in order to also move for white. The only change is checking for 'stalemate' situations, as well as first making the move before checking for a mate. Example:



```
    # The black player moves from a rotated position, so we have to
    # 'back rotate' the move before printing it.
    # note that hist[1].move() aka Position.move function will rotate the board.
    black_move = f'{render(119-move[0]) + render(119-move[1])}'
    print("Black move:", black_move)
    hist.append(hist[-1].move(move))

    if score == MATE_UPPER:
        print("Black won")
        return move, -1
```

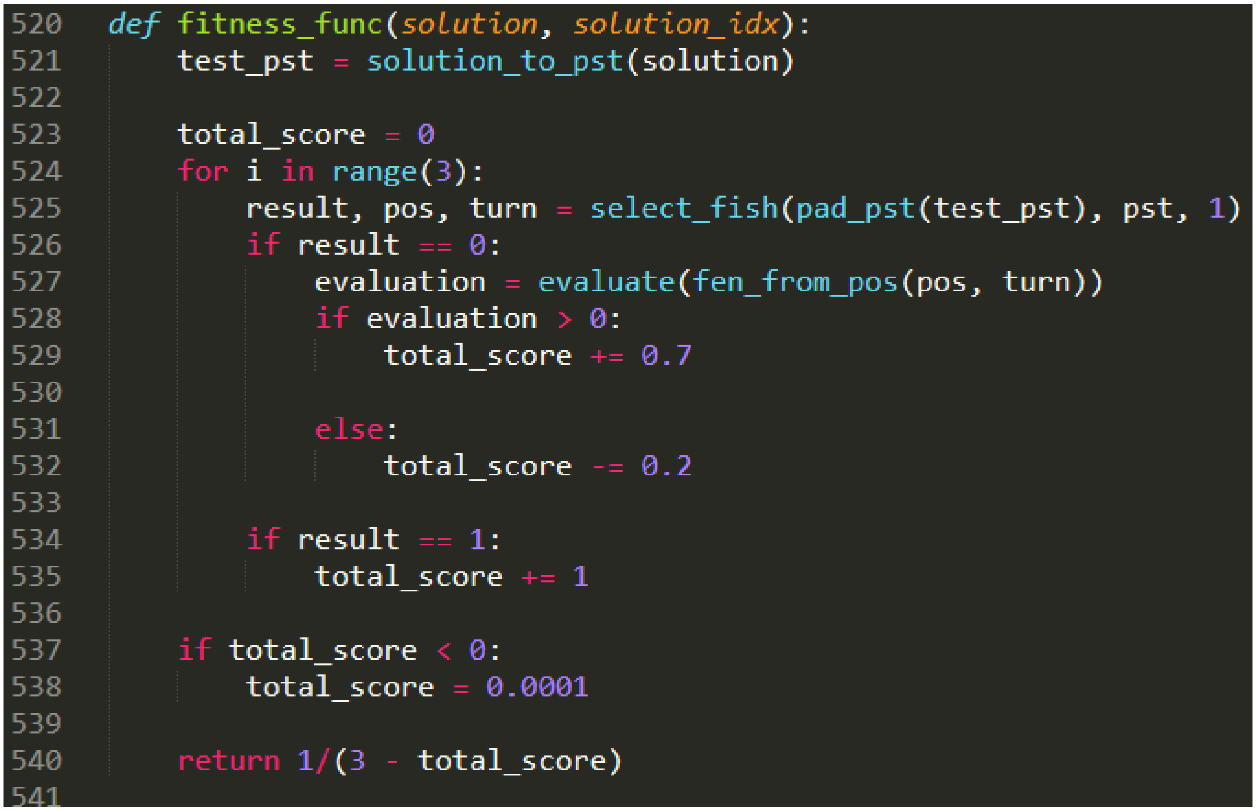
In sunfish, first it checks the score and prints, then makes the move. We reverse this to prevent a situation where we return a value before the move is actually made.

Figure 1: Sunfish main method



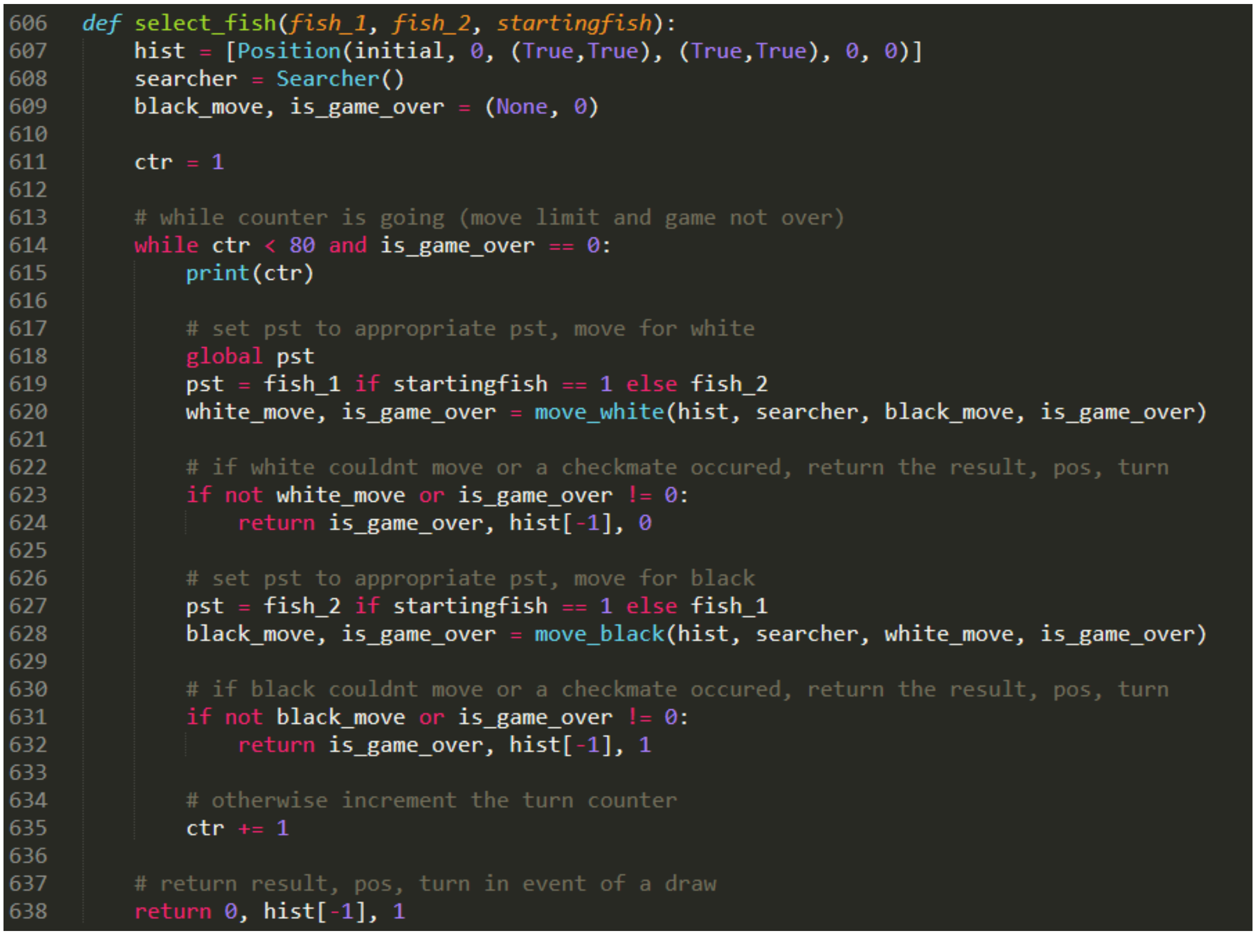
```python
def main():
    hist = [Position(initial, 0, (True,True), (True,True), 0, 0)]
    searcher = Searcher()
    while True:
        # print_pos(hist[-1])

        if hist[-1].score <= -MATE_LOWER:
            print("You lost")
            break

        # We query the user until she enters a (pseudo) legal move.
        move = None
        while move not in hist[-1].gen_moves():
            match = re.match('([a-h][1-8])'*2, input('Your move: '))
            if match:
                move = parse(match.group(1)), parse(match.group(2))
            else:
                # Inform the user when invalid input (e.g. "help") is ent
                print("Please enter a move like g8f6")
        hist.append(hist[-1].move(move))

        # After our move we rotate the board and print it again.
        # This allows us to see the effect of our move.
        print_pos(hist[-1].rotate())

        if hist[-1].score <= -MATE_LOWER:
            print("You won")
            break

        # Fire up the engine to look for a move.
        start = time.time()
        for _depth, move, score in searcher.search(hist[-1], hist):
            if time.time() - start > 1:
                break

        if score == MATE_UPPER:
            print("Checkmate!")

        # The black player moves from a rotated position, so we have to
        # 'back rotate' the move before printing it.
        print("My move:", render(119-move[0]) + render(119-move[1]))
        hist.append(hist[-1].move(move))
```

Figure 2: fitness function



```python
def fitness_func(solution, solution_idx):
    test_pst = solution_to_pst(solution)

    total_score = 0
    for i in range(3):
        result, pos, turn = select_fish(pad_pst(test_pst), pst, 1)
        if result == 0:
            evaluation = evaluate(fen_from_pos(pos, turn))
            if evaluation > 0:
                total_score += 0.7

            else:
                total_score -= 0.2

        if result == 1:
            total_score += 1

    if total_score < 0:
        total_score = 0.0001

    return 1/(3 - total_score)
```

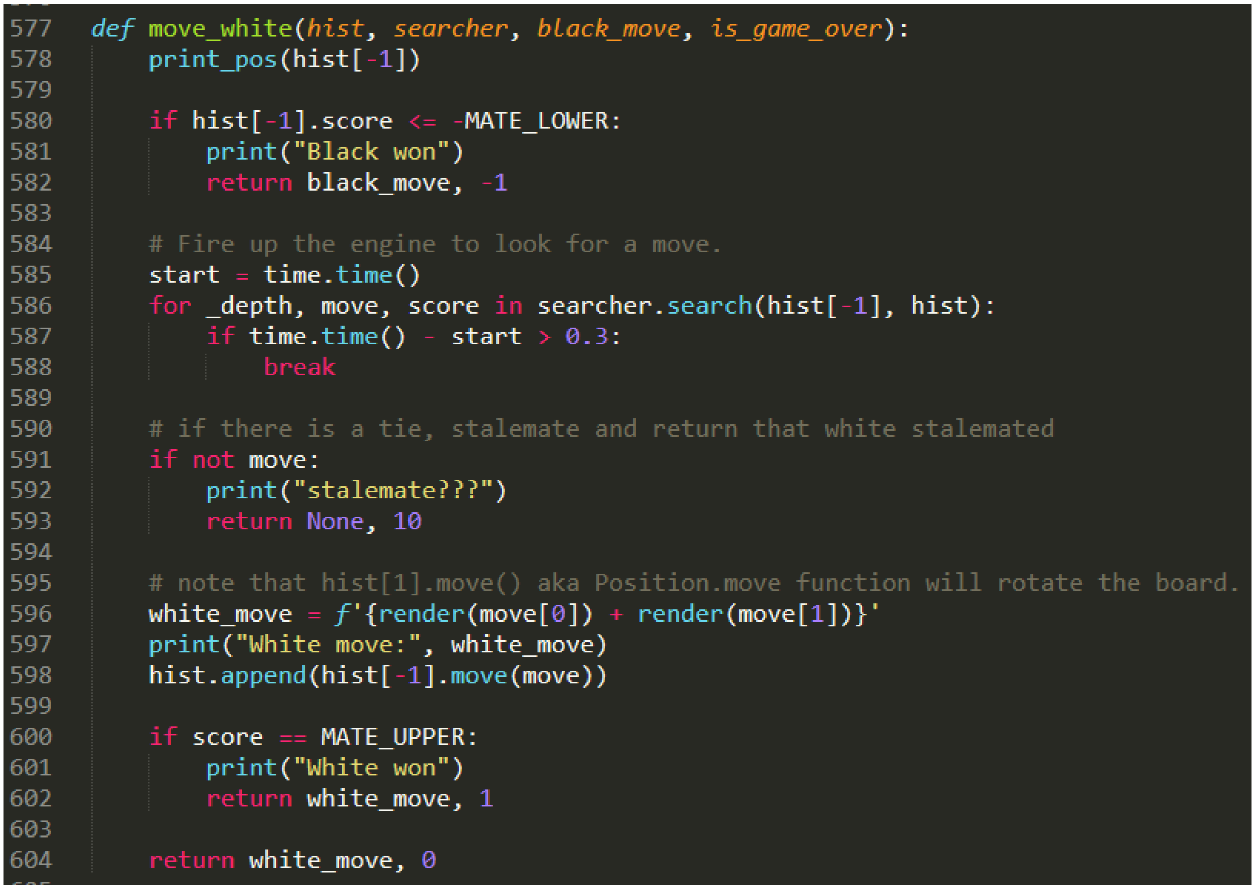
Figure 3: Select_Fish function



```python
def select_fish(fish_1, fish_2, startingfish):
    hist = [Position(initial, 0, (True,True), (True,True), 0, 0)]
    searcher = Searcher()
    black_move, is_game_over = (None, 0)

    ctr = 1

    # while counter is going (move limit and game not over)
    while ctr < 80 and is_game_over == 0:
        print(ctr)

        # set pst to appropriate pst, move for white
        global pst
        pst = fish_1 if startingfish == 1 else fish_2
        white_move, is_game_over = move_white(hist, searcher, black_move, is_game_over)

        # if white couldnt move or a checkmate occured, return the result, pos, turn
        if not white_move or is_game_over != 0:
            return is_game_over, hist[-1], 0

        # set pst to appropriate pst, move for black
        pst = fish_2 if startingfish == 1 else fish_1
        black_move, is_game_over = move_black(hist, searcher, white_move, is_game_over)

        # if black couldnt move or a checkmate occured, return the result, pos, turn
        if not black_move or is_game_over != 0:
            return is_game_over, hist[-1], 1

        # otherwise increment the turn counter
        ctr += 1

    # return result, pos, turn in event of a draw
    return 0, hist[-1], 1
```

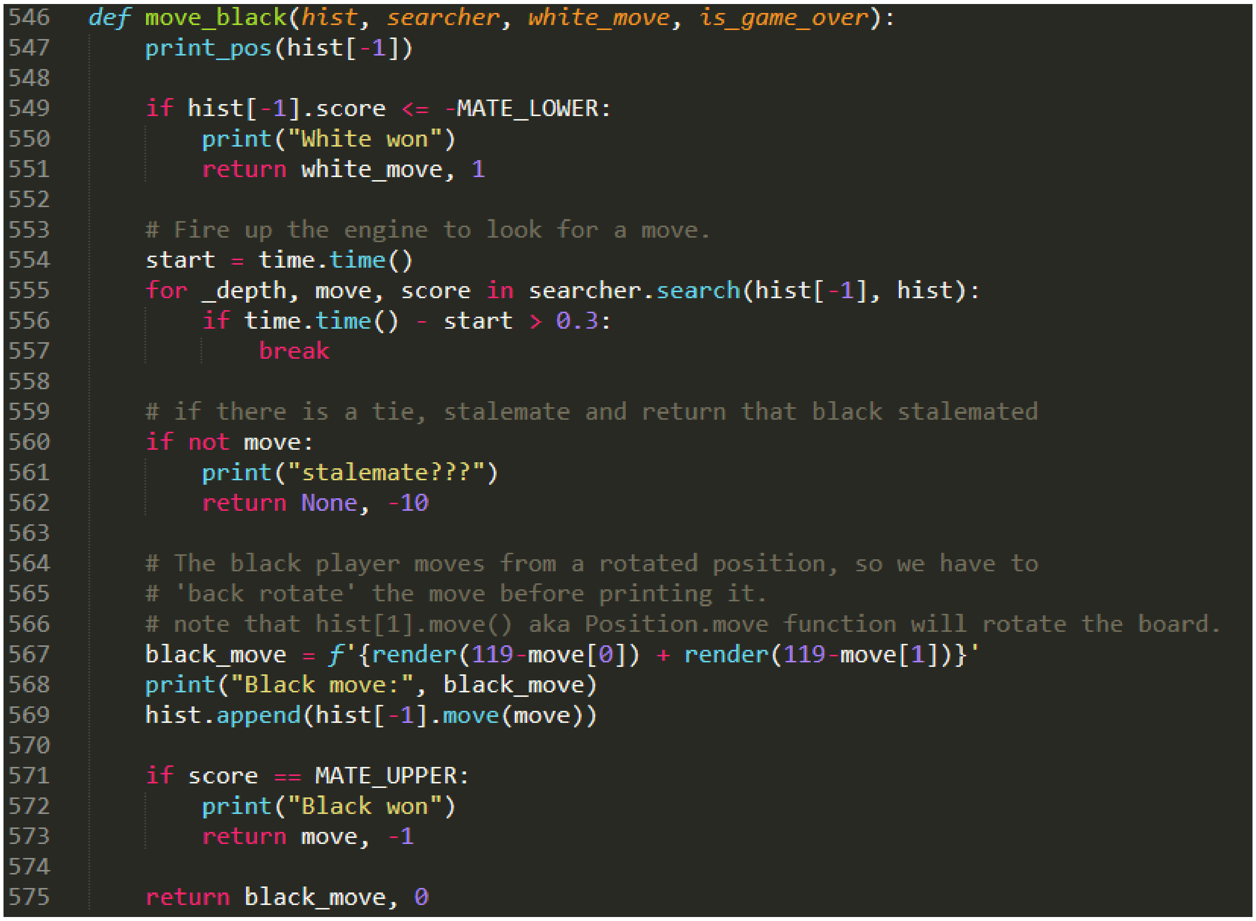
Figure 4: Movement functions:

Figure 4a



```python
def move_white(hist, searcher, black_move, is_game_over):
    print_pos(hist[-1])

    if hist[-1].score <= -MATE_LOWER:
        print("Black won")
        return black_move, -1

    # Fire up the engine to look for a move.
    start = time.time()
    for _depth, move, score in searcher.search(hist[-1], hist):
        if time.time() - start > 0.3:
            break

    # if there is a tie, stalemate and return that white stalemated
    if not move:
        print("stalemate???")
        return None, 10

    # note that hist[1].move() aka Position.move function will rotate the board.
    white_move = f'{render(move[0]) + render(move[1])}'
    print("White move:", white_move)
    hist.append(hist[-1].move(move))

    if score == MATE_UPPER:
        print("White won")
        return white_move, 1

    return white_move, 0
```

Figure 4b



```python
def move_black(hist, searcher, white_move, is_game_over):
    print_pos(hist[-1])

    if hist[-1].score <= -MATE_LOWER:
        print("White won")
        return white_move, 1

    # Fire up the engine to look for a move.
    start = time.time()
    for _depth, move, score in searcher.search(hist[-1], hist):
        if time.time() - start > 0.3:
            break

    # if there is a tie, stalemate and return that black stalemated
    if not move:
        print("stalemate???")
        return None, -10

    # The black player moves from a rotated position, so we have to
    # 'back rotate' the move before printing it.
    # note that hist[1].move() aka Position.move function will rotate the board.
    black_move = f'{render(119-move[0]) + render(119-move[1])}'
    print("Black move:", black_move)
    hist.append(hist[-1].move(move))

    if score == MATE_UPPER:
        print("Black won")
        return move, -1

    return black_move, 0
```