Chess

Carter Kruse (October 5, 2023)

Instructions

In this project, we will write a program for playing chess. There is provided code to serve as a basis for the solution. First, you should run `test_chess.py`. You (presumably a human) will go first, as the white player. You can type a move into the bottom console. The second player is a computer-controlled "A.I.", which cleverly picks a random move and makes it. (I hope you win the first game.)

We will use a Python chess package. It is primarily used to represent board states, find transitions, and make moves. To install it, use the `pip3` command as described in the documentation for the library.

Experimental GUI - Provided is a simple display function based on PyQt with the chess library (with a `.svg` output function) in `gui_chess.py`. To make it work, you will need PyQt. To install it, use `homebrew`, as described on the main course webpage (under installation instructions). You do not need to use the GUI if you are happy with the text output from `test_chess.py`.

Be aware the design of `gui_chess.py` is simple, but works imperfectly. Specifically, the Qt library is event-driven. This means that Qt runs a main event loop, with which you may register events like mouse clicks, timers, or keyboard input. In response to one of those events, Qt will call a callback function that you specify. Since Qt is running on a single thread, Qt will stop responding to input while it waits for that callback function to return. As a consequence, callback functions must execute quickly. Chess AIs are not quick... the UI will freeze during computation. The fi is to create a separate thread to run the AI in. This is not a good use of our time, so for now, we can exit the GUI with `[CTRL] + C` (or using the hard stop button in PyCharm).

Use The Source, Luke

Once you have played the game to exhaustion, read the provided code carefully. This is not a paragon of coding excellence, but read it anyway, as it is helpful to be familiar with it (to complete the assignment). `RandomAI.py` serves as a basic example to get started.

Python Chess Documentation/Source

Read the Python chess documentation and source. The documentation is sparse, though it is sufficient to complete the assignment. Notice that there is an undo move capability, using `push()` and `pop()` on boards. That is important, as creating a new object takes time and memory. It is much

more efficient if the game tree search makes moves and then undoes those moves after exploring them, so that only a single board object is instantiated ever.

Minimax & Cutoff Test

Begin with implementing depth-limited minimax search. The code should follow the pseudo-code for the algorithm presented in the book. The minimax search should stop searching provided some `cutoff_test()` method returns true because one of the following has happened.

- We have reached a terminal state (a win or a draw).
- We have reached the specified maximum depth.

Vary the maximum depth to get a feeling of the speed of the algorithm. Further, have the program print the number of calls it made to minimax, as well as the maximum depth. Record the observations in the document.

The depth-limited minimax search code directly follows from the pseudo-code for the algorithm presented in the book. As indicated, the algorithm stops searching provided some `cutoff_test()` method returns true because we have either reached a terminal state (a win or a draw), or we have reached the specified maximum depth.

The minimax search algorithm is implemented in `MinimaxAI.py` (in Python). While the file contains quite a few comments, we will quickly review the main components/structure here.

The main algorithm creates a list of possible legal moves, randomizes this, and then determines the initial values for the search algorithm. By cycling through the moves, we update the board, and consider which player is making the move. With this information, we either make a call to `max_value()` or `min_value()`, which recursively analyzes the board state to determine the best course of action. With this information, we update the best move/value.

```
  def minimax(self, board):
      # Set the current depth and value equal to 0.
      current_depth = 0
      current_value = 0

      # Determine the set of legal moves from the board, and shuffle for randomization.
      moves = list(board.legal_moves)
      random.shuffle(moves)

      # Set the best value equal to the boundary, and the best move equal to a random move.
      best_value = float('-inf') if board.turn == chess.WHITE else float('inf')
      best_move = moves[0]
```

```
        # Set the maximum depth equal to the instance variable.
        max_depth = self.depth

        # Update the number of minimax moves.
        self.moves += 1

        # Cycle through the possible legal moves.
        for move in moves:
            # Update the state of the board.
            board.push(move)
            self.calls += 1

            # If the "new" turn is the white player, i.e. the move is the black player.
            if board.turn == chess.WHITE:
                # Begin the recursive algorithm.
                current_value = self.max_value(board, current_depth, max_depth)

                # Check to see if the current value beats the best value.
                if current_value < best_value:
                    # Update the variables accordingly.
                    best_value = current_value
                    best_move = move

            # Otherwise if the "new" turn is the black player, i.e. the move is the white pla
            else:
                # Begin the recursive algorithm.
                current_value = self.min_value(board, current_depth, max_depth)

                # Check to see if the current value beats the best value.
                if current_value > best_value:
                    # Update the variables accordingly.
                    best_value = current_value
                    best_move = move

            # Return the board to it's previous state.
            board.pop()

        return best_move
```

The `max_value()` method initally checks to see if the cutoff conditions are satisfied, before creeating a list of possible legal moves and randomizing them. As before, the initial values for the search are determined. By cycling through the moves, we update the board, and with this information, we update the maximum value accordingly, considering the results of `min_value()`, which recursively analyzes the board state to determine the best course of action.

```
    def max_value(self, board, current_depth, max_depth):
        # Check if the cutoff conditions are satisfied.
```

```
        if self.cutoff_test(board, current_depth, max_depth):
            # Return the "utility" of the board position.
            return self.utility(board)

        # Determine the set of legal moves from the board, and shuffle for randomization.
        value = float('-inf')
        moves = list(board.legal_moves)
        random.shuffle(moves)

        # Cycle through the possible legal moves, and apply the recursive algorithm.
        for move in moves:
            board.push(move)
            self.calls += 1
            value = max(value, self.min_value(board, current_depth + 1, max_depth))
            board.pop()

        return value
```

The `min_value()` method initally checks to see if the cutoff conditions are satisfied, before creeating a list of possible legal moves and randomizing them. As before, the initial values for the search are determined. By cycling through the moves, we update the board, and with this information, we update the minimum value accordingly, considering the results of `max_value()`, which recursively analyzes the board state to determine the best course of action.

```
    def min_value(self, board, current_depth, max_depth):
        # Check if the cutoff conditions are satisfied.
        if self.cutoff_test(board, current_depth, max_depth):
            # Return the "utility" of the board position.
            return self.utility(board)

        # Determine the set of legal moves from the board, and shuffle for randomization.
        value = float('inf')
        moves = list(board.legal_moves)
        random.shuffle(moves)

        # Cycle through the possible legal moves, and apply the recursive algorithm.
        for move in moves:
            board.push(move)
            self.calls += 1
            value = min(value, self.max_value(board, current_depth + 1, max_depth))
            board.pop()

        return value
```

The `cutoff_test()` function considers the state of the board, the current depth, and the maximum depth allowed to determine if the search should be stopped for a terminal state (win/draw) or depth

limit.

```
def cutoff_test(self, board, current_depth, max_depth):
    # The search stops if we have reached a terminal state (win/draw)
        # OR we have reached the specified maximum depth.
    if board.is_checkmate() or board.is_stalemate() or current_depth >= max_depth:
        return True
    return False
```

The `utility()` function is responsible for evaluating the "utility" of the board after the cutoff test is applying, so we consider the case where the board is in checkmate, stalemate, or otherwise (in which case an evaluation function is necessary).

```
def utility(self, board):
    if board.is_checkmate():
        # Check if checkmate is against the white player.
        if board.turn == chess.WHITE:
            return float('-inf')

        # Check if checkmate is against the white player.
        else:
            return float('inf')

    elif board.is_stalemate():
        return 0

    else:
        return float(self.evaluate(board))
```

In varying the maximum depth, the following observations are considered. In all cases, the minimax search algorithm performs well against a random AI, as there is an n-move look ahead.

- Maximum Depth = 1 - The algorithm runs extremely quickly, and typically results in around 30-40 player moves and plenty of calls to minimax.
- Maximum Depth = 2 - The algorithm runs relatively quickly, and typically results in around 20-30 player moves and even more calls to minimax.
- Maximum Depth = 3 - The algorithm runs much more slowly, and typically results in around 15-25 player moves and even more calls to minimax.

Evaluation Function

Since we are cutting off the search, we will not always reach a terminal state. We need a heurisitic evaluation function. Consider a material value heuristic (similar to that described by the textbook). Describe the evaluation function used in the documentation.

As indicated, since we are cutting off the search, we will not always reach a terminal state, so we need a heurisitic evaluation function. In this case, the material value heurisitc (similar to that described by the textbook and here) is used.

This evaluation function determines the relative difference in pieces between the white and black players, and uses this information, along with the appropriate weighting system, to determine a rough heuristic for the board state. While this is not the best heurisitic, it serves it's purpose as a way to evaluate who is "winning" in a chess match.

```
# Evaluate
def evaluate(self, board):
    white_pawn, black_pawn = len(board.pieces(chess.PAWN, chess.WHITE)), len(board.pieces
    white_knight, black_knight = len(board.pieces(chess.KNIGHT, chess.WHITE)), len(board.
    white_bishop, black_bishop = len(board.pieces(chess.BISHOP, chess.WHITE)), len(board.
    white_rook, black_rook = len(board.pieces(chess.ROOK, chess.WHITE)), len(board.pieces
    white_queen, black_queen = len(board.pieces(chess.QUEEN, chess.WHITE)), len(board.pie
    white_king, black_king = len(board.pieces(chess.KING, chess.WHITE)), len(board.pieces

    return (white_pawn - black_pawn) + (3 * (white_knight - black_knight)) + (3 * (white_
        (5 * (white_rook - black_rook)) + (9 * (white_queen - black_queen)) + (200 * (whi
```

Once the evaluation function has been implemented, play the computer again, looking for signs of intelligent play. With sufficient depth and allowed time, the program should not lose stupidly and should always take a win. It should not be easy to beat. Use the fact that we may input any initial state to check for this. The computer should try to block wins and take wins. Vary the allowed depth, and discuss this in the documentation.

With the evaluation function implemented, there are certainly signs of intelligent play, considering that with sufficient depth and allowed time, the program does not lose stupidly and show always take a win. In testing, this is not particularly easy to beat. As previously indicated, in varing the maximum depth, the following observations are considered. In all cases, the minimax search algorithm performs well against a random AI, as there is an n-move look ahead.

- Maximum Depth = 1 - The algorithm runs extremely quickly, and typically results in around 30-40 player moves and plenty of calls to minimax.

- Maximum Depth = 2 - The algorithm runs relatively quickly, and typically results in around 20-30 player moves and even more calls to minimax.
- Maximum Depth = 3 - The algorithm runs much more slowly, and typically results in around 15-25 player moves and even more calls to minimax.

Iterative Deepening

A good chess program should be able to give a reasonable move at any time requested. A good approach to such "anytime planning" is to use iterative deepening on the game tree. Once the depth-limited minimax is working, implement iterative deepening. At each depth, the best move might be saved in an instance variable `best_move`. Verify that for some start states, `best_move` changes (and hopefully improves) as deeper levels are searched.

As indicated, if we want to create a good chess program that is able to give a reasonable move at any time requested, we may use iterative deepening on the game tree, specifically the depth-limited minimax. At each depth, the best move is saved to a variable `best_move`, and for start states (depending on the amount of time alloted), the `best_move` changes (and improves) as deeper levels are searched.

The iterative deepening algorithm is implemented in `IterativeDeepeningAI.py` (in Python). While the file contains quite a few comments, it mostly has the same main components/structure as `Minimax.py`, and thus, will not be reviewed in depth.

In varying the maximum depth, the following observations are considered. In all cases, the iterative deepening search algorithm performs well against a random AI, as there is an n-move look ahead.

- Maximum Depth = 1 - The algorithm runs extremely quickly, and typically results in around 30-40 player moves and plenty of calls to iterative deepening.
  - This is equivalent to just running the minimax search algorithm for a maximum depth of 1.
- Maximum Depth = 2 - The algorithm runs relatively quickly, and typically results in around 30-50 player moves and even more calls to iterative deepening.
  - This is equivalent to just running the minimax search algorithm for a maximum depth of 1, along with a maximum depth of 2, hence the increased number of moves/calls.
- Maximum Depth = 3 - The algorithm runs much more slowly, and typically results in around 40-70 player moves and even more calls to iterative deepening.
  - This is equivalent to just running the minimax search algorithm for a maximum depth of 1, along with a maximum depth of 2, along with a maximum depth of 3, hence the increased number of moves/calls.

Alpha-Beta Pruning

Write `AlphaBetaAI.py`, likely by copying the code from `MinimaxAI.py` and extending appropriately. Now, play with the maximum depth and the number of states to see if you are able to search deeper.

As indicated, the alpha beta algorithm is implemented in `AlphaBetaAI.py` (in Python) and is simply an extension of the minimax algorithm implemented in `MinimaxAI.py`. While the file contains quite a few comments, it mostly has the same main components/structure as `Minimax.py`, and thus, will not be reviewed in depth.

In varying the maximum depth, it definitely seems as though we are able to search deeper with the appropriate pruning. This is because when we search at the same depth using minimax and alpha beta, the results are the same, though the execution is faster with alpha beta. This is directly demonstrated by comparing the number of "calls" made to the minimax versus alpha beta algorithm. To see this, please run the provided code. The "time" further gives an indication of the speed of the alpha beta algorithm in comparision to the minimax algorithm.

Thus, we are able to increase the maximum depth for alpha beta, as alpha beta is less computationally heavy, which allows for a further "look-ahead" algorithm to compute the best move according to the heuristic. In this case, as with most adversarial search algorithms, greater depth results in a better outcome.

Add some simple move-reordering and see if this helps you search deeper (or make fewer calls to the alpha-beta function when searching to a specified depth). Record the observations in the documentation.

To further enhance the alpha beta algorithm, allowing us to search deeper (essentially, by making fewer calls to the alpha beta function when searching to a specified depth), we are able to implement simple move-reordering. In the following, we simply prioritize moves that capture a piece on the board, and by using this pre-processing, we are able to speed up the alpha beta algorithm.

```
  def ordered_moves(self, board):
      # Determine the set of legal moves/captures from the board.
      moves = list(board.legal_moves)
      captures = list(board.generate_legal_captures())

      # Create a list of "non-capture" moves that do not capture a piece.
      non_captures = []

      # Cycle through the moves to construct the list.
      for move in moves:
```

```
            if move not in captures:
                non_captures.append(move)

        # Randomize the lists separately to allow for unique movement.
        random.shuffle(captures)
        random.shuffle(non_captures)

        return captures + non_captures
```

In varying the maximum depth, the following observations are considered. In all cases, the alpha beta algorithm performs well against a random AI, as there is an n-move look ahead. Further, the alpha beta algorithm allows us to run a deeper search, and thus performs well against minimax and iterative deepening, as the computational time is speed up for the same depth. As indicated previously, this is demonstrated through the use of the enumeration of the calls, along with the actual time taken to compute a move, both of which are displayed in testing.

The challenge here is testing. There are many examples of code (written in the past) where minimax outplays alpha-beta, because alpha-beta was implemented incorrectly. Test both at the same depth. Ensure that given the same initial position, each returns a move with exactly the same value. If they do not, there is an error. Show the results (briefly) in the documentation, demonstrating that for the same depth, for various positions, alpha-beta explored fewer nodes and yet gave a move (leading to a position satisfying the cutoff test) with the same value.

As indicated, testing both minimax and alpha beta at the same depth indicates that the alpha beta algorithm is implemented appropriately. Given the same initial position, each algorithm returns a move with exactly the same value, though the moves may be different due to the randomization of the moves. In other words, if there are two opportunities to capture a pawn, the algorithms may capture different pawns, though this is simply a result of using a materialistic heuristic, which weighs each situation equivalently.

To demonstrate that for the same depth, for various positions, alpha beta explored fewer nodes and yet gave a move (leading to a position satisfying the cutoff test) with the same value, we may simply observe the number of calls (which represents the number of explored nodes) when running the two algorithms against each other. Alternatively, let us consider the case where we run each algorithm against a random AI, and compare the results, as follows.

The checkmate conditions after running the algorithms for the cases are provided as follows:

`player1 = RandomAI()` and `player2 = MinimaxAI(2)` : Minimax AI Recommended Move: h4e1 (Moves: 21, Calls: 427687, Max Depth: 2) (Time: 0.507 Seconds)

`player1 = RandomAI()` and `player2 = AlphaBetaAI(2)` : Alpha-Beta AI Recommended Move: c6a5 (Moves: 21, Calls: 95382, Max Depth: 2) (Time: 0.198 Seconds)

From this simple example, we can see that because of the maximum depth being the same, the number of moves until checkmate remains relatively the same, though the number of calls for minimax is significantly greater than that for alpha beta. Similarly, the time required for a single move computation in alpha beta is significantly less than that for minimax.

Transposition Table

Implement a transposition table, which requires a hash function. Be careful. Calling a hash on board objects always returns the same value (by observation). You could try something like this as the hash function: `hash(str(game.board))` . A Python set, frozen-set, or dictionary uses the hash method of an object to compute the needed hash value. We might need to write a class that wraps game board and has such a method in order to use the built-in set data structure.

Demonstrate that the code uses the transposition table to prevent some calls in minimax or alpha-beta, and discuss the number of calls made in the documentation.

The alpha beta algorithm with a transposition table included is implemented in `AlphaBetaAI_Transposition.py` (in Python) and is simply an extension of the alpha beta algorithm implemented in `AlphaBetaAI.py` . While the file contains quite a few comments, it mostly has the same main components/structure as `AlphaBetaAI.py` , and thus, will not be reviewed in depth.

In implementing the transposition table with a hash function (which is not on the board object itself, but rather a string form, like `hash(str(board))`), the efficiency of the alpha beta algorithm is enhanced. As indicated, a Python dictionary uses the hash method of an object to compute the needed hash value, though in this case, we do not need to write a class that wraps the game board and has a method in order to use the built-in set data structure. Instead, we simply build the `lookup()` and `store()` within the class we have already developed, as these are relatively simple to create. The parameters already contain the elements of the class itself (`self`) and the `board` , which gives the state of the game.

In varying the maximum depth, it definitely seems as though we are able to search deeper with the transposition table. This is because when we search at the same depth using alpha beta and alpha beta with the transposition table, the results are the same, though the execution is faster with the transposition table. This is directly demonstrated by comparing the number of "calls" made in each of the algorithms. To see this, please run the provided code. The "time" further gives an indication of the speed of the algorithms in comparision to one another.

As previously, we are able to increase the maximum depth for alpha beta with a transposition table, as it is less computationally heavy, which allows for a further "look-ahead" algorithm to compute the best move according to the heuristic. In this case, as with most adversarial search algorithms, greater depth results in a better outcome.

To demonstrate that for the same depth, for various positions, alpha beta with a transposition table explored fewer nodes and yet gave a move (leading to a position satisfying the cutoff test) with the same value, we may simply observe the number of calls (which represents the number of explored nodes) when running the two algorithms against each other. Alternatively, let us consider the case where we run each algorithm against a random AI, and compare the results, as follows.

The checkmate conditions after running the algorithms for the cases are provided as follows:

`player1 = RandomAI()` and `player2 = AlphaBetaAI(2)` : Alpha-Beta AI Recommended Move: e4e1 (Moves: 26, Calls: 111355, Max Depth: 2) (Time: 0.253 Seconds)

`player1 = RandomAI()` and `player2 = AlphaBetaAI_Transposition(2)` : Alpha-Beta AI Recommended Move: g2g1n (Moves: 24, Calls: 60540, Max Depth: 2) (Time: 0.235 Seconds)

From this simple example, we can see that because of the maximum depth being the same, the number of moves until checkmate remains relatively the same, though the number of calls for alpha beta without the transposition table is significantly greater than that for alpha beta with the transposition table. Similarly, the time required for a single move computation in alpha beta with the transposition table is significantly less than that for alpha beta without the transposition table.

Extra Credit/Bonus

The following implementations are designed to enhance the problem, thus meriting "extra credit" as defined by the outlines of the assignment. By going above and beyond to consider extensions to the problem, we consider a greater level of complexity for the problem. Due to the structure of our code, we are able to maintain much of the structure of the previously written code.

Zobrist Hashing - Bonus #1

Zobrist hash functions are not discussed in the text, but are quite interesting and very useful for rapid incremental construction of hash values as the game tree is searched. Implement Zobrist hashign and a transposition table.

This extension allows for implementation of a Zobrist hash function that quickly creates hashes using XOR. The rest of the code for alpha beta remains the same, as we are simply changing the `lookup()`

and `store()` functions, specifically the associated hash for a given board state. This deviates from the typical hashing used by Python, when we hash the string representation of the chess board.

As indicated, the main modification is to the alpha beta Python file. The initialization function is modified to enumerate the pieces and colors, creating a Zobrist table for a given chess board square, according to the color and piece type.

```
def __init__(self, depth):
    self.depth = depth
    self.moves = 0
    self.calls = 0
    self.table = {}

    # Zobrist Hashing
    self.zobrist_table = {}

    # Enumerating the pieces in the chess match.
    pieces = [chess.PAWN, chess.KNIGHT, chess.BISHOP, chess.ROOK, chess.QUEEN, chess.KING

    # Enumerating the colors in the chess match.
    colors = [chess.WHITE, chess.BLACK]

    # Cycle through each of the colors, pieces, and squares on the chess board.
    for color in colors:
        for piece in pieces:
            for square in range(64):
                self.zobrist_table[(color, piece, square)] = random.randint(0, 2**64 - 1)
```

The Zobrist hashing function (which depends strictly on XOR), is provided as follows:

```
# Zobrist Hash (Algorithm)
    # XOR Hashes (Pieces On Squares)
def zobrist_hash(self, board):
    hash = 0

    # Cycle through the squares on the chess board.
    for square in range(64):
        # Determine the piece at a particular square.
        piece = board.piece_at(square)

        # Updating the Zobrist hash according to the piece color, type, and location.
        if piece:
            hash ^= self.zobrist_table[(piece.color, piece.piece_type, square)]

        # Updating the Zobrist hash according to which player's turn it is.
        if board.turn == chess.BLACK:
```

```
                hash ^= 2**64 - 1

        return hash
```

As indicated, we are able to determine the piece at a particular space on the board, and XOR this with the Zobrist table. The hash must also take into consideration whose turn it is, rather than just the state of the board. This completes the hashing function.

Opening Book - Bonus #2

One significant weakness of the program is that it won't play well at the beginning of the game. The material value of the board only changes with a capture, and captures are rare at the beginning. Most chess programs (and human beings) solve this by knowing standard openings; there are books of these standard, called "opening books". Add an opening book.

This extension allows for implementation of an opening book, with various opening series of moves. To do so, we leave most of the code as it is, and update the `test_chess.py` file to create a new `test_chess_openings.py` file. This new file either serves to implement a random opening for the particular game, or simply provides a visual indication of all of the openings created.

As indicated, the main modification is to the testing file. The following openings are provided:

```
opening_book = {'vienna_game': ['e2e4', 'e7e5', 'b1c3'],
                'scotch_game': ['e2e4', 'e7e5', 'g1f3', 'b8c6', 'd2d4'],
                'italian_game': ['e2e4', 'e7e5', 'g1f3', 'b8c6', 'b1c3'],
                'ruy_lopez': ['e2e4', 'e7e5', 'g1f3', 'b8c6', 'b1c3', 'a7a6'],
                'sicilian_defense': ['e2e4', 'c7c5'],
                'french_defense': ['e2e4', 'e7e6'],
                'caro-kann_defense': ['e2e4', 'c7c6'],
                'pirc_defense': ['e2e4', 'd7d6', 'g1f3', 'g8f6'],
                'queens_gambit': ['d2d4', 'd7d5', 'c2c4'],
                'slav_defense': ['d2d4', 'd7d5', 'c2c4', 'c7c6'],
                'kings_indian_defense': ['d2d4', 'g8f6', 'c2c4', 'g7g6'],
                'grunfeld_defense': ['d2d4', 'g8f6', 'c2c4', 'g7g6', 'b1c3', 'd7d5'],
                'english_opening': ['c2c4'],
                'reti_opening': ['g1f3'],
                'giuoco_piano': ['e2e4', 'e7e5', 'g1f3', 'b8c6', 'b1c3', 'g8f6', 'd2d4',
                'two_knights_defense': ['e2e4', 'e7e5', 'g1f3', 'b8c6', 'b1c3', 'g8f6',
                'queens_gambit_declined': ['d2d4', 'd7d5', 'c2c4', 'e7e6', 'b1c3', 'g8f6
                'sicilian_najdorf': ['e2e4', 'c7c5', 'g1f3', 'd7d6', 'd2d4', 'c5d4', 'f3
                }
```

To choose a random opening from this list, and then play the chess game with the two "players" (typically using minimax or alpha beta algorithms), this section of the code should be uncommented:

```
# Choose a random opening.
opening_name = random.choice(list(opening_book.keys()))

# Run the opening book, keeping track of how long it takes (short!).
start = time.time()
for move in opening_book[opening_name]:
    game.board.push(chess.Move.from_uci(move))
end = time.time()

# Provide the name of the opening book, along with the total time taken and result.
print("Opening Book: " + opening_name)
print('(Time: {:.3g} Seconds)'.format(end - start))
print(game)

# Run the chess game, allowing for moves to be made.
while not game.is_game_over():
    print(game)
    game.make_move()

print()
print("WHITE - BLACK")
print(game.board.result())
print()

# print(hash(str(game.board)))
```

As indicated, this chooses a random opening from the "book" of possible openings and applies it to the chess board before the algorithms come into play. To display that the opening "book" actually works, however, we implement the following, which provides a visual representation for all of the openings we have provided:

```
# Cycle through all of the openings in the opening book to demonstrate.
for opening_name in list(opening_book.keys()):
    # Run the opening book, keeping track of how long it takes (short!).
    start = time.time()
    for move in opening_book[opening_name]:
        game.board.push(chess.Move.from_uci(move))
    end = time.time()

    # Provide the name of the opening book, along with the total time taken and result.
    print("Opening Book: " + opening_name)
    print('(Time: {:.3g} Seconds)'.format(end - start))
```

```
    # Ensure that the next opening book starts with the white player.
    game.board.turn = chess.WHITE
    print(game)

    # Reset the chess board.
    game.board.reset()
```

This allows us to see the various openings that are included, which may be used when implementing minimax, iterative deepening, or alpha beta.

Profiling - Bonus #3

Amdahl's Law suggests that it's usually best to improve the things that are slowest. Profile your AI and report the results. Where is most of the execution time being spent? Where might it be improved, either by reducing the number of calls (through more clever heuristics), or by improving the speed of the code (for example, the heuristic function)?

Yes, it seems certainly the case that it is usually best to improve the things that are the slowest. Thus, we profile the various AIs used in this assignment, and report the results below, focusing on where most of the execution time is being spent and where it might be improved, either by reducing the number of calls (through more clever heuristics), or by improving the speed of the code.

MinimaxAI.py Profiling demonstrates that the majority of time is spent in the `minimax()` method and the recursive `max_value()` and `min_value()` calls. This is expected as these methods for the core recursive search. The `choose_move` method takes a negligible fraction of time, as it just calls `minimax()` once and prints the output, so not optimization is needed here.

Within `minimax()`, most time is spent evaluating moves by calling `max_value()` and `min_value()`. We could reduce time in the following ways:

- Implementing alpha-beta pruning to cut off branches that can't improve the result. This reduces the number of recursive calls.
- Caching/memoizing results of `max_value()` and `min_value()` to avoid re-exploring the same branch.

Further, the `utility()` method which evaluates board positions is called many times. We could speed this up in the following ways:

- Pre-computing board evaluation heuristics instead of calculating each time.
- Approximating the evaluation function with a neural network which can evaluate board positions quickly.

In summary, the main optimizations would be to reduce the number of recursive calls to `max_value()` and `min_value()` using techniques that we implement later in the assignment.

AlphaBetaAI.py Profiling demonstrates that the majority of time is still spent in the `minimax()` method and the recursive `max_value()` and `min_value()` calls. This is expected as these methods for the core recursive search.

The new `ordered_moves()` method was added to return moves in a prioritized order, with captures followed by non-captures, which slightly improves ordering.

As indicated previously, the key change between `MinimaxAI.py` and `AlphaBetaAI.py` is the addition of alpha-beta pruning logic in `max_value()` and `min_value()` . This avoids searching branches that cannot improve the final result.

To further optimize the results, we may do the following:

- Add quiescence search to continue searching captures and checks at leaf nodes to avoid horizon effects.
- Cache evaluation results and move ordering heuristics.
- Add move ordering heuristics like killer moves to try more promising moves first. This prunes more branches earlier.

In summary, the key optimizations are alpha-beta pruning to reduce branch search, plus enhanced move ordering, caching, and evaluation functions. Evidently, from the drastic improvement from `Minimax.py` , the pruning has the greatest impact on improving performance.

Enhanced Heurisitic (Mobility) - Bonus #4

This extension allows for the implementation of an enhanced heuristic that does not strictly rely on the material value of the pieces on the board. While the materialistic heuristic functions effectively, a heuristic that considers the structure/layout of the board further proves useful. To implement a heurisitic that takes into consideration the "mobility" of each piece on the board, that is the number of moves that are available, we modify `MinimaxAI.py` slightly to create a new file: `MinimaxAI_Mobility.py` .

While this change is applied to the evaluation function of the minimax algorithm, it could be easily applied to that for alpha beta, as the evaluation functions are consistent between the algorithms. As indicated, the main modification is to the evaluation function, which now includes a mobility term, as follows:

```
  def evaluate(self, board):
      white_pawn, black_pawn = len(board.pieces(chess.PAWN, chess.WHITE)), len(board.pieces
      white_knight, black_knight = len(board.pieces(chess.KNIGHT, chess.WHITE)), len(board.
```

```
    white_bishop, black_bishop = len(board.pieces(chess.BISHOP, chess.WHITE)), len(board.
    white_rook, black_rook = len(board.pieces(chess.ROOK, chess.WHITE)), len(board.pieces
    white_queen, black_queen = len(board.pieces(chess.QUEEN, chess.WHITE)), len(board.pie
    white_king, black_king = len(board.pieces(chess.KING, chess.WHITE)), len(board.pieces

    mobility = self.evaluate_mobility(board)

    return (white_pawn - black_pawn) + (3 * (white_knight - black_knight)) + (3 * (white_
        (5 * (white_rook - black_rook)) + (9 * (white_queen - black_queen)) + (200 * (whi
```

This mobility term represents a positive value if the white pieces have greater mobility than the black pieces, and a negative value if the black pieces have a greater mobility than the white pieces. This is to remain consistent with the "adversarial" component of the heuristic/evaluation function, which is what we did using the materialistic version.

The `evaluate_mobility()` function, with comments, is provided as follows:

```
def evaluate_mobility(self, board):
    # Given the adversarial nature of chess, mobility should be split between players.
    white_mobility, black_mobility = 0, 0

    # Cycle through the squares and pieces located on the board.
    for square, piece in board.piece_map().items():
        # If the piece color is white...
        if piece.color == chess.WHITE:
            # Determine the number of moves that the given piece can take.
            moves = board.attacks(square) & board.occupied_co[chess.WHITE]

            if piece.piece_type == chess.PAWN:
                white_mobility += len(moves) * 0.1
            elif piece.piece_type == chess.KNIGHT:
                white_mobility += len(moves) * 0.3
            elif piece.piece_type == chess.BISHOP:
                white_mobility += len(moves) * 0.5
            elif piece.piece_type == chess.ROOK:
                white_mobility += len(moves) * 1
            elif piece.piece_type == chess.QUEEN:
                white_mobility += len(moves) * 1.5

        # If the piece color is black...
        else:
            # Determine the number of moves that the given piece can take.
            moves = board.attacks(square) & board.occupied_co[chess.BLACK]

            if piece.piece_type == chess.PAWN:
                black_mobility += len(moves) * 0.1
            elif piece.piece_type == chess.KNIGHT:
```

```
                black_mobility += len(moves) * 0.3
            elif piece.piece_type == chess.BISHOP:
                black_mobility += len(moves) * 0.5
            elif piece.piece_type == chess.ROOK:
                black_mobility += len(moves) * 1
            elif piece.piece_type == chess.QUEEN:
                black_mobility += len(moves) * 1.5

    return white_mobility - black_mobility
```

In this case, the weights applied to the mobility are relatively small, considering that there could be a significant number of moves to make for any given piece. Further, the weights seem to reflect the relative importance of the pieces, though this could certianly be modified and further explored. The chosen weights seem to accurately reflect the amount of "regularization" we hope to induce when considering the mobility of the pieces on a chess board.

Literature Review - Bonus #5

The Chess AI problem from this assignment is a well-known one, and it has been studied extensively in the AI research community (alongside problems involving Go and other games). While there is no required additional part of the assignment, I found it relevant to complete a brief review of a somewhat-recent paper on Chess AIs.

Thus, a paper was chosen and read through enough to get a sense of the approach and major findings. In this report, we briefly discuss the paper. The discussion should describe the problem attacked by the paper, give a quick summary of the main result(s), and discuss the basic approach of the paper.

Paper: Chess AI: Competing Paradigms for Machine Intelligence by Shiva Maharaj, Nick Polson, Alex Turk

This paper provides an overview of different approaches to building Chess AI systems, focusing primarily on the contract between neural networks and tree search (minimax, alpha beta) methods.

Specifically, the following quotes highlight the differences:

- "Stockfish uses the alpha-beta pruning search algorithm (Edwards and Hart, 1961). Alpha-beta pruning improves minimax search (Wiener, 1948; von Neumann, 1928) by avoiding variations that will never be reached in optimal play because either player will redirect the game."
- "AlphaZero uses the Monte Carlo Tree Search (MCTS) algorithm to identify the best lines via repeated sampling (Silver, Hubert, et al., 2018). In MCTS, node evaluations at the end of prior

> simulations are used to direct future simulations toward the most promising variations. The search depth increases with each simulation."

The key idea is that while neural networks learn meaningful feature representations/value function from game data, they struggle with long-term planning. Contrastingly, tree search methods like Monte Carlo Tree Search (MCTS) excel at long-term planning but rely on handcrafted game logic.

The authors allude to the idea that combining neural networks and tree search offers the best of both worlds, as neural networks can provide learned evaluation functions and priors to guide/prune MCTS. "Stockfish and LCZero represent two competing paradigms in the race to build the best chess engine. The magic of the Stockfish engine is programmed into its search; the magic of LCZero into its evaluation."

In summary, the paper argues that taking advantage of the strengths of neural networks and tree search will be key not only to creating Chess AIs, but to further developing the field of artificial intelligence, by allowing hybrid systems that learn in a human-like intuitive way, while also focusing on strategic planning.