COSC 76: Artificial Intelligence

Programming Assignment 3: Chess AI

Student: Amittai Wekesa (github: @siavava)

Fall 2021

In this programming assignment, you will write a program for playing chess. The learning objective is to implement the adversarial search methods discussed in class.

REQUIRED TASKS

1. Implement Minimax and Cutoff Test

Begin with implementing depth-limited minimax search. The code should look like the pseudo-code for minimax 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.

Minimax Implementation

```
    def minimax(self, board: Board, depth=None):
        """
            Given a board state, calculate the minimax value of that board state.
            :arg `board`: Chess board state.
            :arg `depth`:
        """

        if not depth: depth = self.depth

        # if cutoff point has been reached, return an evaluation of the board state.
        if self.cutoff_test(board, depth):
            return self.evaluate(board)

        # otherwise, if the target is to maximize, return the max_value.
        elif self.maximizing:
            return self.max_value(board, depth)

        # otherwise, return the min_value.
        else:
            return self.min_value(board, depth)

    def max_value(self, board: Board, depth: int):
        """
            Given a board state, finds the maximum value for that state.
        """

        # if cutoff point has been reached, evaluate the value of the board.
        if self.cutoff_test(board, depth):
            value = self.evaluate(board)
            return value

        # otherwise, recursively find the max of min for each next state,
```

```
        # remembering the best outcome.
        else:
            highest_value = -inf
            all_moves = board.legal_moves

            for move in all_moves:
                board.push(move)
                highest_value = max(highest_value, self.min_value(board, depth-1))
                board.pop()

            return highest_value

    def min_value(self, board, depth):
        """
            Given a board state, finds the minimum value for that state.
        """

        # otherwise, if cutoff point has been reached, evaluate the value of the board.
        if self.cutoff_test(board, depth):
            value = self.evaluate(board)
            return value

        # otherwise, recursively find the max of min for each next state,
        # remembering the best outcome.
        else:
            lowest_value = inf
            all_moves = board.legal_moves

            for move in all_moves:
                board.push(move)
                lowest_value = min(lowest_value, self.max_value(board, depth-1))
                board.pop()

            return lowest_value
```

Cutoff Test Given a board state and the current depth, this function determines if the cutoff conditions have been reached, i.e. the depth is zero or the game is over.

```
    def cutoff_test(self, board: Board, depth: int):
        """
            Given a board state and a search depth, determines whether
            the search should go on or the search should be cut off.
            :arg board: a Chess board object.
            :arg depth: integer repsesenting how far deeper the search should go on.

        """
        return (depth == 0) or (board.is_game_over())
```

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

> Minimax runs at an OK pace at depth 2 and wins consistently wins against random player. However, the minimax AI occassionally gets stuck in one search node, taking over 30 seconds to complete it and slowing down the game.

In one test run, it takes a total of 78.864 seconds to checkmate random player in 17 moves, with an average of 4.639 seconds per call, which is pretty slow, despite it making some moves in less than 2 seconds. On the other hand, random AI spent only 16.025 seconds guessing counter-moves. I used Python's `cProfile` module to profile the algorithm, and it made a total of 657 calls to minimax, 656 calls to max_value and 15113 calls to min_value. The match-up between `minimax` and `max_value` makes sense because, with a cutoff of depth 2, Minimax was only exploring the `max_value` at one level (right from `minimax`), but exploring `min_value` at depth 2, which had much more child branches.

Cumulatively, the algorithm spent a total of 78.804 seconds in `max_value`, but 78.253 of those seconds were spent in `min_value` calls from inside `max_value`.

At depth 3, the `minimax` algorithm moves at a slow pace and takes an upward of 10 seconds per move round.

Profile Stats

```
   Ordered by: cumulative time

  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      33    0.000    0.000   94.891    2.875 /workspace/personal/python/cs76/PA3/ChessGame.py:21(make_
      17    0.003    0.000   78.864    4.639 /workspace/personal/python/cs76/PA3/MinimaxAI.py:42(choos
     657    0.001    0.000   78.837    0.120 /workspace/personal/python/cs76/PA3/MinimaxAI.py:105(mini
     656    0.685    0.000   78.804    0.120 /workspace/personal/python/cs76/PA3/MinimaxAI.py:126(max_
   15113    1.501    0.000   78.253    0.005 /workspace/personal/python/cs76/PA3/MinimaxAI.py:149(min_
```

2. Implement Evaluation function

For the evaluation function, I wenet with a rudimentary interpretation of the game: pawns are value 1, knights and bishops are value 3, rooks are value 5, and queens are value 9.

I also implemented a terminal state analysis that, given an ended game, determines the desirability of the ending for **white** – -INFINITY vs 0 vs INFINITY if white lost, drew, or won, respectively.

```
    def evaluate(self, board: Board):
        """
            Evaluate a Chess position and determine its disirability.
        """

        # if the game is over, return infinity, neg infinity, or zero
        # depending on whether the game has been won, lost, or drawn.
        if board.is_game_over():
            return self.end_status(board)

        # if the game is not yet over, parse the pieces on the board
        # to determine the value of the value of the state.
        white = self.parse_color(board, chess.WHITE)
        black = self.parse_color(board, chess.BLACK)
        return white - black

    @staticmethod
    def end_status(board: Board):
        """
            Determine the desirability of a game end-state.
        """

        # if game is not yet over, print error message and return 0.
        if not board.is_game_over():
            log_error("Game is not over yet.")
            return 0

        result: str = board.outcome().result()
        if result == "1-0":
            return inf
        elif result == "1/2-1/2":
            return 0
        else:
            return -inf

    @staticmethod
    def parse_color(board: Board, suit):
        """
            Given a board state and a suit, parses the pieces of that suit
            on othe board and returns their total value.
        """

        # sum up the total value of the pieces of given suit on the board.
        val = len(board.pieces(chess.PAWN, suit))           # Pawns -> value 1
        val += 3 * len(board.pieces(chess.KNIGHT, suit))    # Knights -> value 3
        val += 3 * len(board.pieces(chess.BISHOP, suit))    # Bishops -> value 3
        val += 5 * len(board.pieces(chess.ROOK, suit))      # Rooks -> value 5
```

```
        val += 9 * len(board.pieces(chess.QUEEN, suit))     # Queens -> value 9

        return val
```

This heuristic is important, as I wanted to be able to get my AI to play as *Black* by simply flipping the minimax initial call for `max_value` to `min_value` and not having to write an alternate evaluation function.

Here are two endings with Minimax playing as White, and as Black. In the White case, the heuristic is maximized, but in the Black case, the heuristic is minimized.

```
#1: WHITE
White to move

First move found, score = 0.
Better move found, score shift from 0 to 38.
Better move found, score shift from 38 to inf.

Minimax AI recommending move = g4g8, move score = inf

. k . . . . Q .
. . . . . . . R
. . P . . P . .
P . . P . . . .
. B . . P P . .
. . . . . . . .
. . . . . . P .
R N . . K B N .
---------------
a b c d e f g h

Black to move

Checkmate? True
Stalemate? False
Number of moves: 22
#2: BLACK

Black to move

First move found, score = -43.
Better move found, score shift from -43 to -inf.

Minimax AI recommending move = d8h4, move score = -inf

r n b . k . n r
. . . p . p p p
. . . . . . . .
p . . . . P . .
. . . . . . q q
b . P . p . . .
. . . . . . . .
. . . . . . . K
---------------
a b c d e f g h
```

```
White to move

Checkmate? True
Stalemate? False
Number of moves: 19
```

I finally tried pitting Minimax against itself playing as Black and white. In most cases, because the white AI was able to see ahead of the game by 1 level (by nature of playing first), it was able to nudge the game into positions where a checkmate became inevitable, but these checkmates occurred after much more moves (average of about 45 to 50), with much less pieces on the board. This is clearly a better outcome for Black than with random AI, which frequently got checkmated by move 20. With a deeper depth, the advantage of being able to see one level ahead would reduce and the ai's should be more evenly matched.

Here is one outcome:

```
Black to move

First move found, score = 18.

Minimax AI recommending move = a4a5, move score = 18

. . . . . .
. . . . . .
. . . . . .
k . . . . .
. . . . . .
N . . . . .
. B K . . .
. . . . Q N .
---------------
a b c d e f g h

White to move

First move found, score = 18.
Better move found, score shift from 18 to inf.

Minimax AI recommending move = f1b5, move score = inf

. . . . . .
. . . . . .
. . . . . .
k Q . . . .
. . . . . .
N . . . . .
. B K . . .
. . . . . N .
---------------
a b c d e f g h

Black to move

Checkmate? True
Stalemate? False
Number of moves: 46
```

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

> The evaluation function returns an estimate on the value of material on the board. This is a good heuristic (better than -1, 0, 1), and does an excellent job of separating search similar states where an absolute win is not imminent but gains or losses of varying magnitudes are incurred. However, it doesn't take into account the relative positions of pieces on the board. A better heuristic would also check the relative positions of pieces and factor that into the utility of the board – for instance, knights are stronger near the center of the board (as they can control more squares), bishops are stronger near or on the long diagonals of the board, especially when the game is open, and rooks tend to be killers on enemy back-ranks. Similarly, a pawn on the $7th$ rank has more value than a pawn on the $2nd$ rank. The challenge would be how to compute all these different sub-heuristics efficiently. As-is, the algorithm is already spending significant time evaluating the board, despite the evaluation barely taking a fraction of a second to run, because of the volume of evaluation calls done in a game. In the chess game with minimax playing itself, a total of 1461657 board evaluation calls were made by the two game engines playing each other, versus only 2559 calls to `minimax`, 23675 calls to `max_value`, and 25229 calls to `min_val`, respectively.

```
   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     2559    0.005    0.000  186.368    0.073 /workspace/personal/python/cs76/PA3/MinimaxAI.py:105(minir
    23675    2.099    0.000  185.550    0.008 /workspace/personal/python/cs76/PA3/MinimaxAI.py:126(max_v
    25229    3.023    0.000  185.260    0.007 /workspace/personal/python/cs76/PA3/MinimaxAI.py:149(min_v
  1461657    2.852    0.000  135.091    0.000 /workspace/personal/python/cs76/PA3/MinimaxAI.py:178(evalu
```

At depth 1, where each ai is simply maximizing its evaluation of the board, the game runs fast and ends is drawn by move 40.

```
Black to move

First move found, score = -6.

Minimax AI recommending move = b1d2, move score = -6

. . b . k . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . n . . . .
. . . . K . . .
----------------
a b c d e f g h

White to move

First move found, score = -6.
Better move found, score shift from -6 to 0.

Minimax AI recommending move = e1d2, move score = 0

. . b . k . . .
. . . . . . . .
```

```
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . K . . .
. . . . . .
----------------
a b c d e f g h

Black to move

Checkmate? False
Stalemate? False
Number of moves: 33
```

Notably, minimax is still able to win against random ai by barely looking at the evaluation functions, despite frequently losing material after losing pieces following moves that gained material but left more valuable pieces exposed.

```
Black to move

Random AI recommending move h5h4
. Q . . Q Q . .
. . . . . .
. . . B . . .
. . . P . . .
. . . . . k
. . . . . P
. . . . . .
. N . . K B N R
----------------
a b c d e f g h

White to move

First move found, score = 46.
Better move found, score shift from 46 to inf.

Minimax AI recommending move = g1f3, move score = inf

. Q . . Q Q . .
. . . . . .
. . . B . . .
. . . P . . .
. . . . . k
. . . . . N . P
. . . . . .
. N . . K B . R
----------------
a b c d e f g h

Black to move

Checkmate? True
Stalemate? False
```

```
Number of moves: 37
```

3. Implement Alpha Beta

My `alpha-beta` implementation is very similar to minimax, with the exception that successive calls to `max_value` and `min_value` perform book-keeping on the highest and lowest values, prune unneeded branches, and in return update these values if needed and pass them to child calls.

For instance, if the current level of the search is a maximizing node, then we know that the parent node is a minimizer and we don't need to keep searching if we encounter a value higher than the lowest value encountered from the parent node. Similarly, the next level of the search will be a minimizer, so they need not bother to keep searching if their current lowest value is lower than the highest value we reported to them (since we're maximizing, we'll just keep our value).

```
    def alpha_beta_search(self, board: Board):
        """
            Given a board state, calculate the minimax value of that board state.
            :arg `board`: Chess board state.
            :arg `depth`:
        """

        # if cutoff point has been reached, return an evaluation of the board state.
        if self.cutoff_test(board, self.depth):
            return self.evaluate(board)

        # otherwise, if the target is to maximize, return the max_value.
        elif self.maximizing:
            return self.max_value(board, self.depth, -inf, inf)

        # otherwise, return the min_value.
        else:
            return self.min_value(board, self.depth, -inf, inf)


    def max_value(self, board, depth, best, worst):
        """
            Given a board state, finds the maximum value for that state.
        """

        # if cutoff point has been reached, evaluate the state of the board.
        if self.cutoff_test(board, depth):
            value = self.evaluate(board)
            return value

        # otherwise, recursively find the max of min for each next state,
        # remembering the state that gives the best outcome.
        #
        # NOTE: If the value is greater than or equal to the worst value from the other search nodes,
        # since we know the next player will be minimizing (and we are maximizing here),
        # we can prune the remaining searches because they are insignificant.
        #
        # otherwise, we:
        #   1. save the value to the transposition table,
        #   2. update the best value seen yet,
        #   3. and continue the looping search on other next states.
        else:
```

```
            highest_value = -inf

            for move in board.legal_moves:
                board.push(move)
                highest_value = max(highest_value, self.min_value(board, depth-1, best, worst))
                board.pop()

                if highest_value >= worst:
                    self.pruned_branches += 1
                    return highest_value

                else:
                    best = max(best, highest_value)

            return highest_value

    def min_value(self, board, depth, best, worst):
        """
            Given a board state, finds the minimum value for that state.
        """

        # if cutoff point has been reached, evaluate the value of the board.
        if self.cutoff_test(board, depth):
            value = self.evaluate(board)
            return value

        # otherwise, recursively find the max of min for each next state,
        # remembering the state that gives the best outcome.
        #
        # NOTE: If the value is less than or equal to the best value from the other search nodes,
        # since we know the next player will be maximizing (and we are minimizing here),
        # we can prune the remaining searches because they are insignificant.
        #
        # otherwise, we:
        #   1. save the value to the transposition table,
        #   2. update the worst value seen yet,
        #   3. and continue the looping search on other next states.
        else:
            lowest_value = inf

            for move in board.legal_moves:
                board.push(move)
                lowest_value = min(lowest_value, self.max_value(board, depth-1, best, worst))
                board.pop()
                if lowest_value <= best:
                    self.pruned_branches += 1
                    return lowest_value

                else:
                    worst = min(worst, lowest_value)

            return lowest_value
```

Discussion of basic algorithm and pruning Using basic pruning, the algorithm was able to easily search 1 to 2 depths deeper than `minimax`, with comparable results.

The ability to search deeper consistently allowed the algorithm to find better moves than the singular move minimax would find, or one reassignment.

```
White to move

First move found, score = 26.
Better move found, score shift from 26 to 27.
Better move found, score shift from 27 to 34.
Better move found, score shift from 34 to 35.

Alpha-Beta AI recommending move = d4d5, move score = 35

. . . Q . B r .
. . . . p . . p
p . n . . . . .
. . . P . p . .
P P k . . . . .
. . . . . . . .
. . . . P P P P
R N . Q K B N R
---------------
a b c d e f g h
```

Sometimes the algorithm was also able to devise a checkmate faster, but on average most games lasted about 2 or three moves less than `minimax`'s games when playing random AI.

```
Black to move

Random AI recommending move e7e5
. . . Q . B r .
. . . . . . . .
p . P . . . . .
. . . . p p . p
P P k . . . . .
. . . . . . . .
. . . . P P P P
R N . Q K B N R
---------------
a b c d e f g h

White to move

First move found, score = 37.
Better move found, score shift from 37 to inf.

Alpha-Beta AI recommending move = e2e4, move score = inf

. . . Q . B r .
. . . . . . . .
p . P . . . . .
. . . . p p . p
P P k . P . . .
. . . . . . . .
```

```
. . . . . P P P
R N . Q K B N R
---------------
a b c d e f g h

Black to move

Checkmate? True
Stalemate? False
Number of moves: 16
```

Each call to `alpha_beta_search` ***with depth*** 3 lasted an average of 0.100 seconds, while comparable calls to `minimax` ***with depth*** 2 lasted 0.120 per call.

At depth 4, `alpha_beta` takes comparabke time to `minimax`'s depth 2 performance. However, the number of better move reassignments didn't seem to change from the previous depth.

In a game that dragged out, lasting 42 moves with `alpha_beta` playing as white, a total of 98021 search prunes were executed (counting each time the algorithm is absconds a search as a single "prune"). The total number of pruned branches is definitely higher, but the book-keeping required to track all those was slowing down the algorithm.

```
Black to move

Random AI recommending move d4e5
Q . . Q . Q . .
. B . . . . . .
. . . . . . . .
. . . P k . . .
. . . . . . . B
. . . . . . . P
R . . N . . . .
. K . . . . N R
---------------
a b c d e f g h

White to move

First move found, score = 42.
Better move found, score shift from 42 to 51.
Better move found, score shift from 51 to inf.
Prune actions (cumulative): 98021

Alpha-Beta AI recommending move = g1f3, move score = inf

Q . . Q . Q . .
. B . . . . . .
. . . . . . . .
. . . P k . . .
. . . . . . . B
. . . . . N . P
R . . N . . . .
. K . . . . . R
---------------
a b c d e f g h

Black to move

Checkmate? True
Stalemate? False
Number of moves: 42
```

Discussion of similarity of results between Minimax and Alpha-Beta To ensure my ALpha-Beta algorithm is correct, I tested it in different positions to ensure it generates moves with the same scores as Minimax.

Test 1

```
./test_chess.py
First move found, score = 0.
Better move found, score shift from 0 to 2.

Minimax AI recommending move = c4d5, move score = 2

Game 1:
 r . . q k b . r
p b . p . . . p
. p . . . . p n
. P . P P . p .
. . . . P . . .
N P . . . N . .
. . . . . P P P
R . . Q K B . R
----------------
a b c d e f g h

Black to move

First move found, score = 0.
Better move found, score shift from 0 to 2.
Prune actions (cumulative): 2064

Alpha-Beta AI recommending move = c4d5, move score = 2

Game 2:
 r . . q k b . r
p b . p . . . p
. p . . . . p n
. P . P P . p .
. . . . P . . .
N P . . . N . .
. . . . . P P P
R . . Q K B . R
----------------
a b c d e f g h

Black to move
```

Test 2

```
./test_chess.py
First move found, score = 8.

Minimax AI recommending move = g2g4, move score = 8

Game 1:
 r . . . k b . r
p . . p . . . p
. p . . . . p n
. P . . P . q .
. . P . b . P .
N P Q . . . . .
. . . . . P . P
R . . . K B . R
----------------
a b c d e f g h

Black to move

First move found, score = 8.
Prune actions (cumulative): 2575

Alpha-Beta AI recommending move = g2g4, move score = 8

Game 2:
 r . . . k b . r
p . . p . . . p
. p . . . . p n
. P . . P . q .
. . P . b . P .
N P Q . . . . .
. . . . . P . P
R . . . K B . R
----------------
a b c d e f g h

Black to move
```

Test 3

```
First move found, score = -7.
Better move found, score shift from -7 to -5.

Minimax AI recommending move = e2f3, move score = -5

Game 1:
 r . . . . r . .
p b . p k . . p
. p . . . . p .
. P . . . . . .
. . P R . . . .
. P . . . B . .
. . . . . P . P
. . . K . . . .
----------------
a b c d e f g h

Black to move

First move found, score = -7.
Better move found, score shift from -7 to -5.
Prune actions (cumulative): 955

Alpha-Beta AI recommending move = e2f3, move score = -5

Game 2:
 r . . . . r . .
p b . p k . . p
. p . . . . p .
. P . . . . . .
. . P R . . . .
. P . . . B . .
. . . . . P . P
. . . K . . . .
----------------
a b c d e f g h

Black to move
```

Test 4

```
./test_chess.py
First move found, score = -6.

Minimax AI recommending move = a3a4, move score = -6

Game 1:
 r . . . . r . .
p . . . . . . p
. p . . . . p .
. P k . . . . .
R . . . . P . .
. . . . . . . .
. . . . . . . P
. . K . . . . .
----------------
a b c d e f g h

Black to move

First move found, score = -6.
Prune actions (cumulative): 652

Alpha-Beta AI recommending move = a3a4, move score = -6

Game 2:
 r . . . . r . .
p . . . . . . p
. p . . . . p .
. P k . . . . .
R . . . . P . .
. . . . . . . .
. . . . . . . P
. . K . . . . .
----------------
a b c d e f g h

Black to move
```

Discussion of move reordering NOTE: In order to avoid corrupting my original alpha-beta version, I implemented this in a separate file.

To implement move reordering, I utilized my priority queue from PA3 to and a new `OrderedMove` abstraction that holds a move and its capacity. This algorithm has other enhancements, including a transposition table and a possible directive to only consider the N moves. To run it without these other enhancements, call the constructor with `memoized=False`

```
################### Added abstraction for reordering moves ##################
class OrderedMove():
    """
        This class represents a search node.
        A search node is a state of the game, along with its value.
    """
    def __init__(self, caller, board: Board, move: chess.Move, max_heap=True):

        self.max_heap: bool = max_heap
        self.move = move

        board.push(move)
        self.value = caller.evaluate(board)
        board.pop()

    def priority(self):
        """Return the current Node's priority value.
            This value is the sum of the transition cost
            to the Node and the heuristic estimating distance
            to the goal state.
        """
        return self.value

    # Comparators for the heapq module.
    # These functions are used ot devise an ordering for AstarNode instances,
    # in order to arrange them in the priority queue.
    #
    # NOTE: My PriorityQueue module was designed to order items using a Min-Heap.
    # To generate a Max-Heap, I reversed the comparators.
    def __gt__(self, other):
        if self.max_heap:
            return self.value < other.value
        else:
            return self.value > other.value

    def __lt__(self, other):
        if self.max_heap:
            return self.value > other.value
        else:
            return self.value < other.value

    def __eq__(self, other):
        return self.value == other.value

    def __ge__(self, other):
        return self > other or self == other
```

```
    def __le__(self, other):
        return self < other or self == other

    def __str__(self):
        return str(self.value)
    ##############################################################################
    ################## Added functionality for reordering moves ##################
    ##############################################################################
    def reorder_moves(self, board: Board, moves: list, max_heap=True):
        """
            Given a board state and a list of legal moves, reorders the moves
            so that the best move is at the front of the list.
            :arg board: Chess board object.
            :arg moves: list of legal moves.
        """

        # Initialize Priority Queue
        ordered_moves = PriorityQueue()

        # For each move, calculate the value of the board *after* the move is made and add it to the qu
        for move in moves:
            ordered_move = OrderedMove(self, board, move, max_heap=max_heap)
            ordered_moves.push(ordered_move)

        return ordered_moves
```

First, each move is loaded into the `OrderedMove` abstraction and added into the priority queue. While considering the "best" moves first seemed to increase the number of pruned branches, the extra work done in reordering the moves appeared to slow down the algorithm and lose the gain. I thought capping the algorithm to, for instance, only considering the 10 or 15 "best" moves might improve performance – and it did.

```
Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    51    0.001    0.000   49.906    0.979 /workspace/personal/python/cs76/PA3/ChessGame.py:21(make_
    25    0.001    0.000   25.040    1.002 /workspace/personal/python/cs76/PA3/RandomAI.py:21(choose
    25   25.032    1.001   25.032    1.001 {built-in method time.sleep}
    26    0.003    0.000   24.861    0.956 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
   251    0.001    0.000   24.548    0.098 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
   250    0.117    0.000   24.530    0.098 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
  2161    0.072    0.000   23.804    0.011 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
  6925    0.399    0.000   20.451    0.003 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
```

Now, random ai took more time guessing moves than the enhanced version of alpha-beta took to generate next moves. While each call to the `alpha_beta_search` algorithm lasts almost the same time (0.100 in the original to 0.098 in the new version), the number of calls reduces because we are only considering a select number of best moves.In total, we only spend 24 seconds in alpha-beta while the call to reorder moves knocks 20 seconds to our performance. As an experiment, I tried aggressively limiting the algorithm to the 3 best moves, and searching deeper depths. That led to even better performance against random moves but was easily outsmarted by the basic alpha-beta which had the advantange of evaluating a wider repertoire of moves.

Additionally, because we are now evaluating less moves, the number of prune actions went down.

Enhanced Alpha Beta, depth 7, move consideration limit of 4 vs random AI

```
   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       53    0.001    0.000  117.805    2.223 /workspace/personal/python/cs76/PA3/ChessGame.py:21(make_
       27    0.002    0.000   91.761    3.399 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
      105    0.000    0.000   91.349    0.870 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
      104    0.319    0.000   91.339    0.878 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
      359    0.248    0.000   91.050    0.254 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
    27371    1.310    0.000   81.222    0.003 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
   644542    1.340    0.000   78.302    0.000 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
   673890    1.346    0.000   64.800    0.000 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
   702338    0.673    0.000   32.971    0.000 /usr/local/lib/python3.9/dist-packages/chess/__init__.py:
   703254    3.001    0.000   32.363    0.000 /usr/local/lib/python3.9/dist-packages/chess/__init__.py:
  1345948    5.199    0.000   31.795    0.000 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
```

Here, the time spent in each call to alpha_beta increases to 0.870 (from 0.100 in the original version), but the total calls to alpha_beta also decrease dramatically to the low 100's. On the other hand, the number of calls to `evaluate` – used by the function that reorder's moves – increases to $673,890$ over the 7 depths, for a game that lasted 27 moves. Clearly, there's an advantage to limiting the number of moves considered, especially when there is some ordering of which moves are ore likely best. However, there is also the problem of not evaluating more moves and missing moves that do not result in immediate material gain but offer better gain after several moves.

Enhanced Alpha Beta (white), depth 7, move consideration limit of 4 vs Alpha Beta (Black), depth 3, no move consideration limit White wins after 96 moves.

```
Black to move

First move found, score = 0.
Prune actions (cumulative): 40446

Alpha-Beta AI recommending move = d5d6, move score = 0

. . . . Q . . .
. . . . . R . .
. . . k . . . .
. K . . . . . .
. . . . . . . P
. . . . . . . .
. . . . . . . .
. . . . . . . .
----------------
a b c d e f g h

White to move

First move found, score = inf.
Pruned 12656 branches.

Enhanced A/B recommending move = f7d7, move score = inf
. . . . Q . . .
. . . R . . . .
. . . k . . . .
. K . . . . . .
. . . . . . . P
```

```
. . . . . .
. . . . . .
. . . . . .
---------------
a b c d e f g h

Black to move

Checkmate? True
Stalemate? False
Number of moves: 96
```

4. Implement Iterative Deepening

I used minimax in my iterative deepening implementation, where, for each call, the iterative deepening polls minimax for the best move, tracking the scores. I also added a quirk in my implementation that, for each depth, the best move from the previous depth is evaluated first so that if its score decreases and another move gets a better score than the new best score (but not necessarily the "best score" from the previous depth), then the algorithm can detect that move A's value went down and move B surpassed it.

```
class IterativeDeepeningAI():


    def __init__(self, max_depth, maximizing=True, timeout=30, debug=False):
        self.maximizing = maximizing
        self.max_depth = max_depth
        self.debug = debug
        self.search_engine: MinimaxAI = MinimaxAI(0, maximizing=maximizing, debug=False)
        self.timeout = timeout
        self.prev_moves = set()

    def choose_move(self, board: Board):
        """
            Given a board state, chooses the best move to play next.
        """

        # check every move and remember the last move that improves the utility.


        best_move, best_cost = None, -inf
        counter = self.timeout


        # iterate over allowed depths.
        for depth in range(1, self.max_depth):

            if best_move:
                board.push(best_move)
                new_cost = self.search_engine.minimax(board, depth=depth)
                board.pop()

                # First, recheck the best move from previous depth and update cost if it changes.
                if new_cost != best_cost:
                    if self.debug:
                        log_error(f"Move {best_move} changed in score from {best_cost} to {new_cost}.")
                    best_cost = new_cost

            # get every possible move and explore it to the current depth.
            for move in board.legal_moves:

                # try the move
                board.push(move)

                # get the utility of the board after the move.
                cost = self.search_engine.minimax(board, depth=depth)

                # check if the move improves the utility.
```

```
                # NOTE: we check whether it *matches* the utility, OR if it *betters* the utility.
                # This helps avoid a repetition loop where a sequence of first-occurring moves loop bac
                # to each other and the game gets stuck in a loop.
                # However, we also need to avoid blindly chosing the last move played --
                # so we check if the state of the board after the move has been recorded before
                # in the prev_moves set.
                if ( (self.maximizing) and (cost >= best_cost) ) \
                    or ( (not self.maximizing) and (cost <= best_cost) ):

                    # if the cost strictly improves the utility, remember it and log progress.
                    if cost != best_cost:

                        # if debug enabled, print progress
                        if self.debug:
                            if not (best_cost == -inf or best_cost == inf):
                                log_debug_info(f"Depth {depth}; better move found, score shift from {bes
                            else:
                                log_debug_info(f"First move found at depth {depth}; score = {cost}.")

                        best_move, best_cost = move, cost

                    elif not ((cost == best_cost) and (str(move) in self.prev_moves)):
                        best_move, best_cost = move, cost


                # undo the move
                board.pop()

                # if the timeout is reached, stop searching other moves. Otherwise, decrement the timeo
                if counter <= 0: break
                else: counter -= 1

            if counter <= 0:
                break

            # if debug is enabled, print progress
            if self.debug:
                log_debug_info(f"Depth {depth}, best move: {best_move}, cost: {best_cost}.")


        # once the best move is found, remember it and return it.
        self.prev_moves.add(str(move))

        # print information on chosen best move.
        log_info(f"Iterative Deepening AI recommends move {best_move} with cost {best_cost}")

        # return chosen move.
        return best_move
```

Instances when the same move improved:

```
Black to move

Random AI recommending move d8d7
r n b . . . n .
```

```
p p . q . . p .
. . . . . k . p
. . p . p . . .
. . . . . . . .
. P . . . . P .
P . P P P P . P
R . B Q K B N R
---------------
a b c d e f g h

White to move

First move found at depth 1; score = 7.
Depth 1, best move: c2c4, cost: 7.
Move c2c4 changed in score from 7 to 15.
Depth 2, best move: c2c4, cost: 15.
Iterative Deepening AI recommends move b3b4 with cost 15
r n b . . . n .
p p . q . . p .
. . . . . k . p
. . p . p . . .
. P . . . . . .
. . . . . . P .
P . P P P P . P
R . B Q K B N R
---------------
a b c d e f g h
```

Instances when a better move was found:

```
Black to move

Random AI recommending move d8f6
r n b . k b n r
. p p p . . p .
p . . . p q . p
. . . . . . . .
. . . . . . . .
. P . . . . P .
P . P P P P . P
R . B Q K B N R
---------------
a b c d e f g h

White to move

First move found at depth 1; score = -2.
Depth 1, best move: c2c4, cost: -2.
Depth 2; better move found, score shift from -2 to 6.
Depth 2, best move: b3b4, cost: 6.
Iterative Deepening AI recommends move b3b4 with cost 6
```

5. Writeup and Discussion

1. Description: How do your implemented algorithms work? What design decisions did you make? **discussed above.**
2. Evaluation: Do your implemented algorithms actually work? How well? If it doesn't work, can you tell why not? What partial successes did you have that deserve partial credit? **All my algorithms worked, as discussed above.**
3. Responses to discussion questions that are included within the points in "The Task". Here the discussion questions are reported for ensuring that you found all of them: **discussed above.**

    1) (minimax and cutoff test) Vary maximum depth to get a feeling of the speed of the algorithm. Also, have the program print the number of calls it made to minimax as well as the maximum depth. Record your observations in your document.

    2) (evaluation function) Describe the evaluation function used and vary the allowed depth, and discuss in your document the results.

    3) (alpha-beta) Record your observations on move-reordering in your document.

    4) (iterative deepening) Verify that for some start states, best_move changes (and hopefully improves) as deeper levels are searched. Discuss the observations in your document.

EXTRA IMPLEMENTATIONS

1. Transposition Table

I implemented a transposition table enabling computed move scores to be stored and retrieved faster when positions are re-encountered. In my tests, the alpha-beta algorithm performed faster and was able to search deeper with the transposition table activated. However, when terminal state computations were computed, the algorithm occassionally lost points in positions that were first found as terminal states and saved, without foresight of what happens afterward. As a result, I prefer not to store the terminal state values.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
    This module implements a Transposition Table.
"""
__author__ = "Amittai"
__copyright__ = "Copyright 2021"
__credits__ = ["Amittai"]
__email__ = "Amittai.J.Wekesa.24@dartmouth.edu"
__github__ = "@siavava"

class TranspositionTable(object):
    def __init__(self):
        self.data: dict = {}
        self.len = 0

    def __bool__(self):
        return self.len != 0

    def __getitem__(self, key):
        return self.data.get(str(key), None)

    def __contains__(self, key):
        """Check if the table contains an item.
        """
        if str(key) in self.data:
            return True

        return False

    def __setitem__(self, key, value):
        self.data[str(key)] = value
        self.len += 1

    def __str__(self):
        return str(self.data)

    def __len__(self):
        return self.len

    # def zobrist_hash(self, board: Board):
    #     """Return the Zobrist hash of a board.
    #     """
    #     hash_value = 0
```

```
    #     for square in range(64):
    #         piece = board.piece_at(square)
    #         if piece:
    #             hash_value ^= self.zobrist_piece_table[piece.piece_type][square]
```

Performance With Transposition Table The memoization of alpha-beta helps avoid repeated evaluation of encountered states. Combined with move reordering and a limit of 7 on the number of considered moves, I was able to get alpha-beta to only take 1.668 seconds per move turn while searching up to a depth of 5, and still checkmate random AI in 15 moves, a comparable move count (if not better than!) the original Minimax AI, which took way more time per move.

```
Black to move

Random AI recommending move h8g8
. . . . k B r .
. P . . . p . p
. . P . p . . .
. . R . . . p .
. P . P . . . .
. . . P . . . .
. . . . . P P P
. N . Q K B N R
----------------
a b c d e f g h

White to move

First move found, score = inf.
Transposition Table size: 11332.
Pruned 2467 branches.
Re-encountered 1024 states (cumulative)

Enhanced A/B recommending move = b7b8q, move score = inf
. Q . . k B r .
. . . . . p . p
. . P . p . . .
. . R . . . p .
. P . P . . . .
. . . P . . . .
. . . . . P P P
. N . Q K B N R
----------------
a b c d e f g h

Black to move

Checkmate? True
Stalemate? False
Number of moves: 15
         33290485 function calls (33279567 primitive calls) in 38.571 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
```

```
        29    0.001    0.000   39.053    1.347 /workspace/personal/python/cs76/PA3/ChessGame.py:21(make_
        15    0.002    0.000   25.027    1.668 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
        58    0.000    0.000   24.809    0.428 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
  8429/58    0.077    0.000   24.804    0.428 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
 2830/283    0.114    0.000   24.422    0.086 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
      3618    0.201    0.000   14.615    0.004 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
    114409    0.242    0.000   14.156    0.000 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
        14    0.001    0.000   14.023    1.002 /workspace/personal/python/cs76/PA3/RandomAI.py:21(choose_
```

2. Profiling

A. Minimax; Search Depth 2 I used Python's cProfile module (found in the standard Python installation, with reference from this online manual).

For my minimax algorithm here were the results.

A lot of time is spent on move evaluation (for frontier moves), despite each call to `evaluate()` barely taking 0.00 seconds. This is understandable since most positions, most of which end up not being chosen, are on the frontier of the search algorithm at some point and have to be evaluated. In fact, a total of 459990 board evaluations were performed over 45 seconds, despite only 35 moves happening in the game. Clearly, the search, while exhaustive, is inefficient. A possible improvement will be to prune the number of frontier positions evaluated.

```
         77974843 function calls (77514856 primitive calls) in 95.685 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       67    0.001    0.000   96.415    1.439 /workspace/personal/python/cs76/PA3/ChessGame.py:21(make_
       34    0.009    0.000   63.360    1.864 /workspace/personal/python/cs76/PA3/MinimaxAI.py:42(choos
     1119    0.003    0.000   63.303    0.057 /workspace/personal/python/cs76/PA3/MinimaxAI.py:105(mini
     1116    0.554    0.000   63.232    0.057 /workspace/personal/python/cs76/PA3/MinimaxAI.py:126(max_
    14599    1.217    0.000   62.602    0.004 /workspace/personal/python/cs76/PA3/MinimaxAI.py:149(min_
   459990    0.958    0.000   45.509    0.000 /workspace/personal/python/cs76/PA3/MinimaxAI.py:178(evalu
       33    0.001    0.000   33.050    1.002 /workspace/personal/python/cs76/PA3/RandomAI.py:21(choose
       33   33.040    1.001   33.040    1.001 {built-in method time.sleep}
```

B. Alpha-Beta; Search Depth 3 I used Python's cProfile module (found in the standard Python installation, with reference from this online manual).

For alpha-beta, a there is a notable improvement in the number of evaluations of frontier positons, but a total of 323949 evaluations over 33 seconds still points to an inefficiency.

```
         56715414 function calls (56357396 primitive calls) in 63.880 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       25    0.000    0.000   64.513    2.581 /workspace/personal/python/cs76/PA3/ChessGame.py:21(make_
       13    0.003    0.000   52.494    4.038 /workspace/personal/python/cs76/PA3/AlphaBetaAI.py:41(cho
      384    0.001    0.000   52.474    0.137 /workspace/personal/python/cs76/PA3/AlphaBetaAI.py:107(al
      383    1.296    0.000   52.452    0.137 /workspace/personal/python/cs76/PA3/AlphaBetaAI.py:127(ma
     8956    0.550    0.000   52.069    0.006 /workspace/personal/python/cs76/PA3/AlphaBetaAI.py:166(min
   323949    0.775    0.000   33.809    0.000 /workspace/personal/python/cs76/PA3/AlphaBetaAI.py:211(eva
   647850    2.960    0.000   17.790    0.000 /workspace/personal/python/cs76/PA3/AlphaBetaAI.py:246(pa
   367815    0.408    0.000   17.525    0.000 /usr/local/lib/python3.9/dist-packages/chess/__init__.py:
   367839    1.869    0.000   17.120    0.000 /usr/local/lib/python3.9/dist-packages/chess/__init__.py:
  1163206    2.342    0.000   12.302    0.000 /usr/local/lib/python3.9/dist-packages/chess/__init__.py:3
       12    0.000    0.000   12.017    1.001 /workspace/personal/python/cs76/PA3/RandomAI.py:21(choose
       12   12.013    1.001   12.013    1.001 {built-in method time.sleep}
```

C. Enhanced Alpha-Beta; Search Depth 5 I used Python's cProfile module (found in the standard Python installation, with reference from this online manual).

For my enhanced alpha-beta algorithm with memoization, move reordering, and forward pruning, the number of move evaluations dropped to 122065 lasting a total of 11.994 seconds, a notable improvement. The change

would have been greater had I not used evaluations for move ordering.

```
         33290485 function calls (33279567 primitive calls) in 38.571 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       29    0.001    0.000   39.053    1.347 /workspace/personal/python/cs76/PA3/ChessGame.py:21(make_
       15    0.002    0.000   25.027    1.668 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
       58    0.000    0.000   24.809    0.428 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
       58    0.077    0.000   24.804    0.428 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
      283    0.114    0.000   24.422    0.086 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
     3618    0.201    0.000   14.615    0.004 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
   114409    0.242    0.000   14.156    0.000 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
       14    0.001    0.000   14.023    1.002 /workspace/personal/python/cs76/PA3/RandomAI.py:21(choose_
       14   14.017    1.001   14.017    1.001 {built-in method time.sleep}
   122065    0.262    0.000   11.994    0.000 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
   114409    0.082    0.000   11.206    0.000 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
    35913    2.518    0.000    7.841    0.000 /usr/local/lib/python3.9/dist-packages/chess/__init__.py:
```

3. Advanced Move Reordering

I implemented a version of IDS, "Better AI", that attempts to use move costs from previous IDS iterations to reorder mvoes in the current iteration. Using it, I was able to perform IDS up to 6 or 7 levels down before it timed out. However, there are tradeoffs in comparison to the enhanced version of alpha-beta – especially since it's impossible to memoize the search engine in IDS since it frequently revisits board states – it'll simply return values for previous depths and skip the current depth. I tried to get around this by creating an alternate transposition table used specifically for move lookk-ups from the IDS algorithm, but the tradeoff from losing the proper memoization of alpha-beta leads to lower performance.

```
Black to move

Random AI recommending move e7e5
. n b q k . n r
r p . p . . b p
. . . . . p P .
p . p . p . P .
P . . . P . . .
. . . . . . . B
. P P P . . . P
R N B Q K . N R
---------------
a b c d e f g h

White to move

First move found at depth 1; score = 2.
Depth 1, best move: g5f6, cost: 2.
Move g5f6 changed in score from 2 to 3.
Depth 2; better move found, score shift from 3 to 7.
Depth 2, best move: g6h7, cost: 7.
Depth 3, best move: g6h7, cost: 7.
Depth 4, best move: g6h7, cost: 7.
Depth 5, best move: g6h7, cost: 7.
Move g6h7 changed in score from 7 to 10.
Depth 6, best move: g6h7, cost: 10.
Better AI recommends move g6h7 with cost 10
. n b q k . n r
r p . p . . b P
. . . . . p . .
p . p . p . P .
P . . . P . . .
. . . . . . . B
. P P P . . . P
R N B Q K . N R
---------------
a b c d e f g h
```

Once again, the thousands of redundant move evaluations creep up, weighing down the algorithm.

```
Better AI recommends move h5e8 with cost inf
. . . N Q . k .
. . . r . . r .
b . . . . . . .
p . . . . B . .
P . . b P . . .
```

```
. . . . . . . .
. P P P . . . P
R N B . K . . R
---------------
a b c d e f g h

Black to move

Checkmate? True
Stalemate? False
Number of moves: 22
         317230759 function calls (317059205 primitive calls) in 298.610 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       43    0.001    0.000  301.728    7.017 /workspace/personal/python/cs76/PA3/ChessGame.py:21(make_
       22    0.010    0.000  280.695   12.759 /workspace/personal/python/cs76/PA3/BetterAI.py:56(choose
      638    0.005    0.000  280.060    0.439 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
108751/627    1.053    0.000  280.013    0.447 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.p
65915/2485    1.337    0.000  277.480    0.112 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.p
    62209    3.291    0.000  242.626    0.004 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
  1754005    3.901    0.000  235.432    0.000 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
  1866605    4.259    0.000  198.408    0.000 /workspace/personal/python/cs76/PA3/EnhancedAlphaBetaAI.py
```

NOTES

I created a short library, erratum, which simply prints out specific text in different colors.

It made it easier for me to distinguish between the text I'm logging as general info, versus debug info, vs error info. However, I noticed that while it works in Visual Studio Code, it doesn't work in PyCharm because, for some reason, my version of Pycharm doesn't process the specific escape wequences I used to format and color text. If working on PyCharm and the output by your editor seems a bit weird, you may wish to replace the logic in the `text_color` function to simply return the text it's given instead of formatting the text. Or better yet, check out VS Code :)

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
    This file implements functionality to print errors, debug messages,
    and specific info in a distinct way to separate it from normal output.

"""
__author__ = "Amittai"
__copyright__ = "Copyright 2021"
__credits__ = ["Amittai"]
__email__ = "Amittai.J.Wekesa.24@dartmouth.edu"
__github__ = "@siavava"


def __text_color(r: int, g: int, b: int, text: str):
    """
        This function is used to generate colored text in the terminal.
        Used internally and not intended to be called directly.
    """
    return f"\033[1m\033[38;2;{r};{g};{b}m{text} \033[38;2;255;255;255m\033[0m"


def log_error(*args):
    """
        Given a list of arguments presumably representing an error message,
        this function prints them in a distinct way to separate them from normal output.
    """
    err_message = ""
    for arg in args:
        err_message += str(arg) + " "

    print(__text_color(220, 20, 60, err_message))

def log_info(*args):
    """
        Given a list of arguments presumably representing useful information,
        this function prints them in a distinct way to separate them from normal output.
    """
    err_message = ""
    for arg in args:
        err_message += str(arg) + " "

    print(__text_color(30, 144, 255, err_message))

def log_debug_info(*args):
    """
        Given a list of arguments presumably representing debug information,
        this function prints them in a distinct way to separate them from normal output.
    """
    debug_message = ""
    for arg in args:
        debug_message += str(arg) + " "
```

```
    print(__text_color(55, 155, 55, debug_message))


if __name__ == "__main__":
    log_error("Hello", "World")
    log_info("Hello", "World")
    log_debug_info("Hello", "World")
```