CS76 Assignment 3 - ChessAI

Himadri Narasimhamurthy - January 24, 2019

Chess AI - Introduction

In this assignment - I implemented simple versions of both Minimax search and AlphaBeta pruning in order to create a chess-playing AI. Much of the work to actually generate and check states and take in user input and display the board was done by the *python-chess* library. In this report, I will focus more on the search algorithms. In order to run the AI and play against it, you can navigate to the test_chess.py file and run the HumanPlayer() against either AlphaBetaAI(depth) or MinimaxAI(depth). This will require you writing in your own player1 and player2 if you would like to test them against each other or against the randomAI engine.

Minimax Search

While begining my implementation of the algorithm, following the pseudocode from the textbook, I noticed that there would already by two ways which I could implement the value getting algorithm.

1. In two separate functions for minimum and maximum
2. In one minimax function which keeps track of which turn we are on

I decided on the second and implemented it as so in minimax(self, max depth, board, max tf):

```
# minimax algorithm to get values of board
def minimax(self, max_depth, board, max_tf):
    .
    .
    .
    if self.cutoff_test(board, max_depth):
        return -1*getMaterialValue(board)
    .
    .
    .
```

This is our recursive condition - until cutoff_test is True, we want to keep searching. When it is true, we return the value of the current board.

```
    .
    .
    .
    #in case of maximizing
    if (max_tf):
        max_move_val = -10000
        for m in move_list:
            move = chess.Move.from_uci(str(m))
            board.push(move)
            #recursive call to maximize value
            max_move_val = max(max_move_val, self.minimax(max_depth - 1, board, not m
ax_tf))
            board.pop()
        return max_move_val

    #in case of minimizing
    else:
        min_move_val = 10000
        for m in move_list:
            move = chess.Move.from_uci(str(m))
            board.push(move)
            #recursive call to minimize value
            min_move_val = min(min_move_val, self.minimax(max_depth - 1, board, not m
ax_tf))
            board.pop()
        return min_move_val
```

Here, is where we did the recursive calls for implementing minimax search. This code works its way down the game tree doing a simple recursive computation to determine min and max values. We set some extremes in order to declare our initial variables and then continue making our min and max comparisons down the tree.

Cutoff_Test

I also create a simple cutoff test function where I check the main conditions for exiting the recursion for minimax. The first, since we want to implement this using depth limiting search, is by checking our max depth. Next, we must check whether we have reached a goal state and the game is over - then we exit recursion. If not, return false and keep recursing.

```
def cutoff_test(self, board, max_depth):
    if max_depth == 0:
        return True
    if max_depth != 0 and board.is_game_over():
        return True
    else:
        return False
```

Testing Minimax

Though I do not know how to play chess, I did a rigorous amount of testing on minimax vs both the randomAI chess engine and the HumanPlayer. The marker that minimax seemed to be working according to the appropriate evaluation is seen below - when white pawn P in d6 approached, the next state should be that any p in c7, d7, or e7 should capture it.

In my first iteration of minimax, this was the bug that I discovered and it was fixed by implementing iterative deepening search - which I will detail soon.

```
    Please enter your move:
        d5d6
    True
    r . b q k b r .
    p p p p p p p p
    n . . P . n . .
    . . . . . . . .
    . . . . . . . .
    . . . . . N . .
    P P P . P P P P
    R N B Q K B . R
    ----------------
    a b c d e f g h

    Black to move

    Minimax ran 60840 times!
    r . b q k b r .
    p p p p . p p p
    n . . p . n . .
    . . . . . . . .
    . . . . . . . .
    . . . . . N . .
    P P P . P P P P
    R N B Q K B . R
    ----------------
    a b c d e f g h

    White to move
```

Evaluation Function

My evaluation function is in heuristics.py since it is used in both Minimax and AlphaBeta and is a fairly simple implementation of the material heuristic from the textbook. First, we have a helper function which takes a piece and returns its material value.

```
def getPieceValue(p):
    .
    .
    .
    return value
```

The second part of this process is the actual evaluation where we sum the piece values across the board. I did

this in the getMaterialValue function where I summed positive or negative values for pieces across the board in order to get our total value. This took some reading in *python-chess*' documentation.

```
def getMaterialValue(board):
    i = 1
    total = 0
    #number 63 from python-chess documentation
    while i < 63:
        if board.piece_at(i) is not None:
            #if white - then we add positive value
            if bool(board.piece_at(i).color):
                add = getPieceValue(str(board.piece_at(i)))

            #if black we add a negative value
            else:
                add = -1*getPieceValue(str(board.piece_at(i)))
            total = total + add

        i = i+ 1
    return total
```

Though I do not have the time, I would like to experiement with combinations of heuristics that I have now read about while looking through ChessAI articles.

Iterative Deepening

To actually get our best move from our search algorithms, we needed to implement iterative deepening search - rather than the previously tested, depth-limiting search (which can often not reach a solution and instead exit on max-depth).

I do this in applyMinMax(...) and applyAB(...) as seen below. First, I get possible board moves and initialize extremes for my "best move" which I want to be returning

```
#uses minimax value to determine best move - iterative deepening
def applyMinMax(self, depth, board, max_t):
    #get all possible moves and set extremes
    possibleMoves = board.legal_moves
    best_val = -10000
    best = None
    .
    .
    .
```

Then, I run the minimax function which recursively searches through the game tree for min and max values for all possible moves - something that could be done quicker if saved in a dictionary. These values returned from minimax(...) are saved in m*val and compared to my best*val and if m*val exceeds best*val, m becomes my best possible move.

```
    .
    .
    .
     #run minimax and compare it with best_val to determine best node
     for m in possibleMoves:
         move = chess.Move.from_uci(str(m))

         #adds each move onto the stack in order to avoid creating new objects
         board.push(move)
         m_val = max(best_val, self.minimax(depth - 1, board, not max_t))
         board.pop()

         if m_val>best_val:
             best_val = m_val
             best = m

     print("Minimax ran " + str(self.depth) + " times!")
     return best
```

Finally, we must return the best move to the game driver which I do in the choose_move function that returns move and iterates until we find a suitable move or until we reach a depth limit of 100.

```
#calls the final iteration of minimax/ab determining move
def choose_move(self, board, depth_limit = 100):
    .
    .
    .
            return move
```

Alpha Beta Pruning

Much of AlphaBetaAI.py was code from Minimax but with additional alpha and beta values and evaluations which ensured that we would be deeply exploring parts of the trees which could actually contain max or min values - not all parts of the game tree. The code for max is seen below - min is similar but beta is set to the min of beta and min*move*val. This ensures that we only go down paths where we can find minimums or maximums.

```
    # have the max val and set as alpha
        alpha = max(alpha, max_move_val)

        beta = min(beta, min_move_val)
```

My main test for alpha beta vs minimax was that at first, we should see the same move since we are running the same start code - then as we continue to play the game, alphabeta should run quicker since it has a smaller tree to explore and it should also reach a goal state quicker.

```
    AlphaBeta ran 1249 times!
    r n b q k b n r
    p p p p p p p p
    . . . . . . . .
    . . . . . . . .
    . . . . . . . .
    . . . . . . . N
    P P P P P P P P
    R N B Q K B . R
    ----------------
    a b c d e f g h

    Black to move

    Minimax ran 9301 times!
    r n b q k b . r
    p p p p p p p p
    . . . . . . . n
    . . . . . . . .
    . . . . . . . .
    . . . . . . . N
    P P P P P P P P
    R N B Q K B . R
    ----------------
    a b c d e f g h

    White to move
```

Transposition Table

Though I did not have the time to properly debug a transposition table for my search algorithms, I see that doing so would have significantly improved the speed - especially of Minimax at higher depths since we would not have to rerun minimax on board conditions we had already visited before. This transposition table should hash in a string of the board and save the value returned from minimax and since lookup in a dictionary is

constant time, this would allow minimax and alphabeta to run quicker since we would have saved state values.