Chess AI Report

Paolo Takagi-Atilano

Introduction

The purpose of this report is to discuss my implementation of a chess engine that makes use of different variations of Minimax, Iterative Deepening, and Alpha-Beta Pruning algorithms. I discuss the advantages and disadvantages of each algorithm and its subsequent variations. Finally I will discuss an article written by Andreas Junghanns which attempts to determine the effectiveness of Alpha-Beta pruning compared to other methods to write chess engines.

Design

A goal of the design was to keep the entire project as modular as possible. Hence, there were essentially 3 different types of `*.py` files that were used. Here is a brief overview:

AI Algorithms:

`HumanPlayer.py` : Contains `HumanPlayer` class that takes user input to determine what move to make.

`RandomAI.py` : Contains `RandomAI` class that chooses a random (legal) move each turn to make.

`MinimaxAI.py` : Contains `MinimaxAI` class that chooses a "best" move by use of Minimax algorithm, with parameter input of heuristic function and depth limit.

`IterativeDeepeningAI.py` : Contains `IterativeDeepeningAI` class that chooses a "best" move by use of Iterative Deepening algorithm, with parameter input of heuristic function and depth limit. Also prints out discovered "best" move at each depth start at 1.

`AlphaBetaAI.py` : Contains `AlphaBetaAI` class that chooses a "best" move by use of Alpha-Beta algorithm, with parameter input of heuristic function and depth limit. Also, contains `ReorderAlphaBetaAI` class that chooses a "best" move by use of Alpha-Beta algorithm with move reordering, with parameter input of heuristic function and depth limit. Finally, contains `TransAlphaBetaAI` class that chooses a "best" move by use of Alpha-Beta algorithm with move reordering and transposition tables, with parameter input of heuristic function and depth limit.

Heuristics:

`heuristics.py` : Contains `STANDARD_VALUES` dictionary that contains a map of the name of each piece (not considering their color and not counting blank pieces) in string format to their material value, according to standard material value systems and with king high enough to be far away the most important piece (pawn - 1, knight - 3, bishop - 3, rook - 5, queen - 9, king - 200). Also contains `std_get_material` heuristic evaluation function that takes in a board and returns the net material value score for whichever player's turn it is.

Testing/Other:

`test_chess.py` : Contains code that will set up chess games aginst two players (humans or computers). Here the starting board is also altered for different tests.

`gui_chess/py` : Code to use a gui to dispaly the chess game. It was not used at all because the assigment recommended not doing so.

Board Testing Starting Positions:

The following board set ups were used for different testing (all are white to move except Board 5 where it is black to move)

Board 1 (starting position):

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

Board 2 (early-game - Sicilian Defense):

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

Board 3 (mid-game):

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

Board 4 (end-game):

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

Board 5 (puzzle):

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

Evaluation Function

The purpose of the below search algorithms is to select what they consider to be the best moves for a given board state. But in order to do so, there must be a method to compare different potential board states. Hence some sort of evaluation function is necessary. For my evaluation function, I chose to use material. This is relatively simple to implement. I found the difference of each type of piece for white and black, and then multiplied that value by the piece's material value (acuquired using a hard-coded dictionary). The king was given a material value of 200. This was to reflect that the king is the most valuable piece in that it determines who wins or loses the game.

Minimax and Cutoff Test

First I implemented Minimax search with a cutoff test to limit depth. The idea of Minimax is that it is a variation of (path-chechking) depth first search that attempts how to show how a game will play out if each player plays optimally (assuming the heuristic is perfect, which of course it is not). So it essentially iterates all the way down the tree until a cutoff is reached (identified by the cutoff test), and then finds the highest or lowest possible value, depending on whose turn it is. Eventually, it has figured out the "ideal" move for the given situation. The cutoff test is faily simple, so just look at the code for it. In theory if Minimax did not use a cutoff test, it would be able to iterate through the entire tree of every possible game of chess. In practice, that takes far to long, so cutoff tests are necesary.

Minimax Algorithm:

I used the pseudocode in the book to implement the Minimax algorithm. Here is a more detailed look at my specific implementation (the `choose_move` function starts this recursive loop by calling `min_value`):

max_value:

```
# simulate max player
def max_value(self, depth, board):

    # increment function calls count
    self.function_calls += 1

    # checks to see if it is a cutoff
    if self.cutoff_test(depth, board):
        return self.heuristic_fn(board)

    # find maximum possible value assuming min plays optimally
    v = float('-inf')
    for move in board.legal_moves:
        board.push(move)
        v = max(v, self.min_value(depth + 1, board))
        board.pop()

    return v
```

min_value:

```
# simulate min player
def min_value(self, depth, board):

    # increment function calls count
    self.function_calls += 1

    # checks to see if it is a cutoff
    if self.cutoff_test(depth, board):
        return self.heuristic_fn(board, self.depth_fix)

    # find minimum possible value assuming max plays optimally
    v = float('inf')
    for move in board.legal_moves:
        board.push(move)
        v = min(v, self.max_value(depth + 1, board))
        board.pop()

    return v
```

As you can see, depths alternate between calling `min_value` and `max_value`, until the cutoff limit has been reached.

Cutoff Test:

The cutoff test checks to see if the depth limit has been reached, or if the game is over (handled by `python-chess`).

```
def cutoff_test(self, depth, board):
    return self.depth <= depth or board.is_game_over()
```

Testing:

I tested the number of function calls at varying depths to set an upper bound for function calls. To do this, I set `MinimaxAI()` as the white player for different boards, and waited for it to make its move:

Board 1:

Depth 0: 20 function calls, 0 value

Depth 1: 420 function calls, 0 value

Depth 2: 9,322 function calls, 0 value

Depth 3: 206,603 function calls, 0 value

A couple of facts that convince me that the algorithm is correct are that Depth 0 has 20 function calls, which is the number of different first moves a player can make, and that the number of function calls increases exponentially.

*Board 3: *

Depth 0: 35 function calls, 3 value

Depth 1: 1,111 function calls, -2 value

Depth 2: 39,894 function calls, 2 value

Depth 3: 1,275,708 fuction calls, -2 value

*Board 4: *

Depth 0: 1 function call, 5 value

Depth 1: 33 function calls, -8 value

Depth 2: 1,016 function calls, 1 value

Depth 3: 25,078 function calls, 0 value

Iterative Deepening

In the real world, chess is played on a clock. That means that the algorithm is not necesarily allowed to run to completion in order to find the "best" move, given a certain heuristic and depth limit. That means that it could be a smart strategy for the chess engine to use anytime planning so that it has an answer at any point in time. A way to implement this is to use iterative deepening. This algorithm is given a depth limit as an input. It then runs the Minimax algorithm from depth 0 to that depth until the maximum depth has been reached. While an emergency cutoff was not implemented, at least now the chess engine always stores a "best move" for practically any point in time (excluding the small fraction of a second at the beginning).

Testing:

I ran the tests on multiple boards, but some were not too interesting because the best move did not change. So below are states where the board changed, and it was easy for me to analyze whether or not the IDS made a better move or not:

Board 2:

Depth 0: d4c6

Depth 1: d4c6

Depth 2: f1b5

Depth 3: f1b5

Here we see the IDS reaching a smarter at deeper depths. Initially it seems no problem with capturing the black knight on c6. But at depth 2, it realizes that knight is actually protected by the pawn on b7, so capturing that knight is actually just a trade. Instead it opts to develop its bishop on square f1 to b5, which attacks the same black knight and pins that knight against the black king on e8, so that it cannot flee. That is obviously a much better move.

Board 5:

Depth 0: h5a5

Depth 1: h5h1

Depth 2: h5a5

Depth 3: h5h1

Here initially, the AI sees the that it can take the opposing queen and opts to do that. But upon looking further

ahead, it realizes that it can force the opponent to give up its king (aka checkmate) at another depth. Hence, it opts for that move, as winning the game is better than taking an enemy's queen. It is interesting to note that on even depths, the algorithm instead opts to take the queen rather than go for the checkmate. I think that this is because the program is not recognizing that the game is a checkmate, and so it is thinking that it is possible that the enemy queen can take the friendly king, which would result in a material advantage for the other side. A potential solution would be to program win and loss utilities of infinite value when they were recognized.

Alpha-Beta Pruning

Alpha-Beta Pruning (ABP) is a more optimal version of Minimax. It only searches nodes that are worthwhile. That means that if a certain path is guarenteed to return values that it knows will either be too good (according to the heuristic) for the opponent to accept, or too bad for it to accept, it is not worth looking any further into that subtree. This means that there is potentially a dramatic reduction in the number of function calls that are made from Alpha Beta compared to Minimax. This speeds up the algorithm significantly, which means that if it is used in the real world, it would be able to iterate through more nodes in the same amount of time. This advantage could potentially lead to victory.

Alpha Beta Algorithm:

The code is very similar to that of Minimax, but with a couple lines to implement the pruning:

max_value:

```
# simulate max player
def max_value(self, depth, board, alpha, beta):

    # increment function calls count
    self.function_calls += 1

    # checks to see if it is a cutoff
    if self.cutoff_test(depth, board):
        return self.heuristic_fn(board)

    # find maximum possible value assuming min plays optimally
    v = float('-inf')
    for move in board.legal_moves:
        board.push(move)
        v = max(v, self.min_value(depth + 1, board, alpha, beta))
        board.pop()

        # pruning
        if v >= beta:
            return v
        alpha = max(alpha, v)

    return v
```

min_value:

```
 # simulate min player
def min_value(self, depth, board, alpha, beta):

    # increment function calls count
    self.function_calls += 1

    # checks to see if it is a cutoff
    if self.cutoff_test(depth, board):
        return self.heuristic_fn(board)

    # find minimum possible value assuming max plays optimally
    v = float('inf')
    for move in board.legal_moves:
        board.push(move)
        v = min(v, self.max_value(depth + 1, board, alpha, beta))
        board.pop()

        # pruning
        if v <= alpha:
            return v
        beta = min(beta, v)

    return v
```

As one can see, the algorithm is very similar to the Minimax algorithm, except that there is pruning at tne end right before the return statemeny. This ensure that if the value does not fit within the alpha beta vounds, it is returned, and otherwise the alpha beta bounds are set to be more rigorous that they can be.

Testing:

In order to test, I ran the exact same tests as I did for the Minimax algorithm. Below are the reslts:

Board 1:

Depth 0: 20 function calls, 0 value

Depth 1: 420 function calls, 0 value

Depth 2: 1,298 function calls, 0 value

Depth 3: 13,873 function calls, 0 value

Board 2:

Depth 0: 35 function calls, 3 value

Depth 1: 1,111 function calls, -2 value

Depth 2: 8,392 function calls, 2 value

Depth 3: 246,987 function calls, -2 value

Board 3:

Depth 0: 1 function call, 5 value

Depth 1: 33 function calls, -8 value

Depth 2: 161 function calls, 1 value

Depth 3: 3,925 function calls, 0 value

There are a couple observations here to make. First, that the quantity of function calls that Alpha-Beta is able to avoid gets much larger as the depth increases. This makes sense because the greater the depth, more of an impact the pruning at higher depths will have. Second, that the values of the Alpha-Beta pruning are the same as the Minimax algorithm, implying that it is indeed a more efficient algorithm for no tradeoff.

Move Reordering (basic)

Alpha-Beta can get lucky. If it find a path of action that forces (in theory) a good path of play, then it can prune off every node that is lower than it. Hence, the order in which the nodes are explored clearly matters. This is where the idea of move reordering is introduced, or a system that encourages the Alpha-Beta algorithm to prioritize potentially "better" nodes in its search.

Move Reordering Implementation:

I judged each move's potential by using the provided heuristic, and then ordered them using a priority queue. Below is further insight into my implementation:

MovePq:

```
class MovePq:
    def __init__(self, move, priority):
        self.move = move
        self.priority = priority

    def __lt__(self, other):
        return self.priority > other.priority
```

reorder:

```
# uses provided heuristic to order moves
def reorder(self, moves_list, board):

    # the empty heap
    ordered = []

    # building the heap
    for move in moves_list:
        board.push(move)
        priority = self.heuristic_fn(board)
        heappush(ordered, MovePq(move, priority))
        board.pop()

    return ordered
```

It is worth noting that the `choose_move` function now has to iterate though `MovePq` objects instead of moves, butt the necessasry trivial changes were made.

Testing:

I ran the same tests that I did for Alpha Beta and Minimax algorithms. The results were the same to that of the Alpha Beta. I have a couple theories as to why. Perhaps the ordering method I used was already implicity used for the Alpha Beta. This seems possible considering that the ordering method I used was the same as the heuristic I used for Alpha Beta. The second possibiliy would be an error in implementation, but it seems unlikely that the error would generate the exact same results as Alpha Beta (not better or worse, or completely different).

Transposition Table

There can be multiple ways to get to certain board states. For example, `1: e4 e5, 2: d4 d5` can also be reached by `1: e4 d5, 2: d4 e5`, `1: d4 d5, 2: e4 e5`, and `1: d4 e5, e4 d5`. All of these will evaluate to the same score by the evlauation function, as the final state is the exact same. Hence, it not efficient for a search algorithm to investigate all ways to reach a board state. Instead it is more efficient to just store the value of that board state the first time it is reached, and all subsequent times the board state is reached, just return the previously found value and not continue exploring further down that pathway in the tree.

In order to implement this, I made a new search algorithm called `TransAlphaBetaAI`.

I decided that it only made sense to implement a transposition table for Alpha Beta Pruning, and not Minimax.

This is because the reason for having transposition tables is to optimize the number of function calls. Since Minimax is less optimal than Alpha-Beta in this respect and returns moves of the same value as Alpha-Beta, then implementing a transposition table for Minimax would be optimizing an inherently sub-optimal algorithm. This does not make sense when we already have a more optimal alternative.

Implementation:

Again, the implementation is very similar to that of the above. The only difference is that the values take in a parameter called `transposition_table` (which is a dictionary created by the `choose_move` function. It then only runs the next step in the recursive max-min loop if the value is not already in the transposition table. One can see we create a string version of the board, which is because all boards hash to the same value. The strings are distinct, as they use FEN notation to indicate the board state.

max_value

```
# simulate max player
def max_value(self, depth, board, alpha, beta, transposition_table):

    # increment function calls count
    self.function_calls += 1

    # check to see if it is a cutoff
    if self.cutoff_test(depth, board):
        return self.heuristic_fn(board)

    # find maximum possible value if min plays optimally
    v = float('-inf')
    for move in board.legal_moves:
        board.push(move)

        # transposition table checks
        board_str = str(board)
        if board_str not in transposition_table.keys():#or transposition_table[board_
str] < v:
            v = max(v, self.min_value(depth + 1, board, alpha, beta, transposition_ta
ble))
            transposition_table[board_str] = v
        else:
            v = transposition_table[board_str]
        board.pop()

        # pruning
        if v >= beta:
            return v
        alpha = max(alpha, v)

    return v
```

min_value:

```
# simulate min player
def min_value(self, depth, board, alpha, beta, transposition_table):

    # increment function calls count
    self.function_calls += 1

    # check to see if it is a cutoff
    if self.cutoff_test(depth, board):
        return self.heuristic_fn(board)

    # find minimum possible value if max plays optimally
    v = float('inf')
    for move in board.legal_moves:
        board.push(move)

        # transposition table checks
        board_str = str(board)
        if board_str not in transposition_table.keys():#or transposition_table[board_
str] < v:
            v = min(v, self.max_value(depth + 1, board, alpha, beta, transposition_ta
ble))
            transposition_table[board_str] = v
        else:
            v = transposition_table[board_str]
        board.pop()

        # pruning
        if v <= alpha:
            return v
        beta = min(beta, v)

    return v
```

Testing:

Board 1:

Depth 0: 20 function calls, 0 value

Depth 1: 420 function calls, 0 value

Depth 2: 1,038 function calls, 0 value

Depth 3: 6,024 function calls, 0 value

Board 3:

Depth 0: 35 function calls, 3 value

Depth 1: 1,110 function calls, -2 value

Depth 2: 8,908 function calls, 1 value

Depth 3: 119,208 function calls, -2 value

Board 4:

Depth 0: 1 function call, 5 value

Depth 1: 33 function calls, -8 value

Depth 2: 158 function calls, 1 value

Depth 3: 2,267 function calls, 0 value

As we can see, the transposition table found the same value as the AlphaBeta algorithm, but needing fewer function calls to get there, making it faster. The one except was for Depth 2 for Board 3, where it found a worse solution (less value), and in more function calls. But besides that, it was strictly better than the unaltered Alpha Beta. Hence, I am not concerned, as that seems to be very much the exception rather than the rule. I tried adding some other statements to resolve this one case, but could not find anything to work.

Opening Book (bonus)

For further exploration, I decided to implmenet an opening book. I decided to do this because I was getting frusturated with how these games would end up in loops of the AI decided to move the rook back and forth or something like that. I thought of a couple solutions to this issue. One solution would be to implement a heuristic which would provide bonuses for controlling more spaces, or the center of the board. But I decided against implementing this because I saw that it could become much more complicated, because I would have to have different heuristics to prioritize different play at different points of the game, and then I would have to figure out good ways to determine which point of the game it was.

So instead, I decided to implement an opening from which the AI religiously follows the first 3 moves from. After that point, the AI returns to using the AI to make its moves. I used the `TransAlphaBeta` AI for the `OpeningBook` AI, because it is the fastest. In order to implement the opening book, I hard programmed two different opening dictionaries, one for white and one for black. I then had the `OpeningBook` AI select a random integer, which would determined which, when coupled with whichever color the AI was playing as, would detemine the opening to be played. The opening would be played till completion (I kept track of number

of moves, and finished after the opening was out of moves to play), and then after the opening was complete it would just feed the board to the included `TransAlphaBeta` AI to determine which move to play.

I included a human player against the `OpeningBook` AI at the end of my testing, and played against it a couple of times. With the aid of a more formal opening, the games became much more exciting, because the AI had some developed pieces, and such it was easier for it to find exciting lines of play to use.

Related Work (bonus)

I chose this article (Are There Practical Alternatives To Alpha-Beta in Computer Chess? by Andreas Junghanns) because so far this report has presented Alpha-Beta (with optimizations) to be the best possible AI for a chess engine. While this is true for the algorithms listed above, this might not be true for every possible algorithm, so I am hoped that this article will shed some light on that.

First the article discusses some problems in Alpha-Beta. Some of these the following: the algorithm assumes that heuristic evaluation of board states is perfect (which cannot be true bcause if it was true search would not be necesary), that Alpha-Beta can spend much of its computation time exploring pathways that are obviously bad moves, that there could be multiple good alternatives (within the bounds of heuristic error) that are ignored due to Alpha-Beta only considering the minimum and maximum values of nodes, that Alpha-Beta has no measure of confidence of for the selected "best" move, and finally that Alpha-Beta assumes that the opponent is using the same heuristic evlauation as itself.

Next, it presents alternative approaches that attempt to solve these problems. There are quite a few, but here are some specific examples:

M&N-Backup Procedure: This procedure uses the M and N best successor values of a node for it alpha and beta values, rather than the max and min. This means the algorithm should prioritize pathways that have more, better choices (than less).

Baynesian Game Tree Search (BP): Uses an evaluation function that returns probability distributions, which are the expected change of the value if the search were to expand that leaf node. Based off of this information, the algorithm (in theory) has more information about which paths are wroth pursuing. Unfortinately, it has been hard ot find reliable eestimators for how deeper searches will change the current value of a position.

Best-First Minimax: An algorithm that traverses the tree up and down. First down to reach a leaf of the node where the current algorithm is, and then back up to evaluate the values.

ProbiMax: An algorithm that is meant to exploit the fact that humans do not play according to a heuristic by playing (perceived) suboptimal moves. This should make it harder for the human opponent to force a draw according to the 50-move rule in chess. This needs an accurate model of the opponent, and it is currently not clear how to obtain accurate approximations to build this model.

In conclusion, it determines that there are no solutions at the moment that are more effective than Alpha-Beta (as many of these algorithms run into their own problems), but there is potential for that to change in the future. Perhaps a reason of Alpha-Beta's ability is because of its simplicity, making the algorithm relatively fast. More computaitonally complicated methods may be "smarter", but also just not able to visit enough nodes to compete against Alpha-Beta. Also, it concluded that while these algorithms may not be superior to Alpha-Beta in chess, some can outperform Alpha-Beta in other two-player, zero-sum, perfect-information games. For example, BP was shown to outperform Alpha-Beta in Othello.