

COSC 76 Project 3: Chess

Seungjae Jason Lee

Introduction

Objective

There are two main objectives to the project. First, we want to implement Minimax search. Then, we want to implement Alpha Beta Pruning. Using these two different searches, we can build an AI that plays chess.

Game Rule

The game rule is basic chess rule.

Code Design/ Building the Model

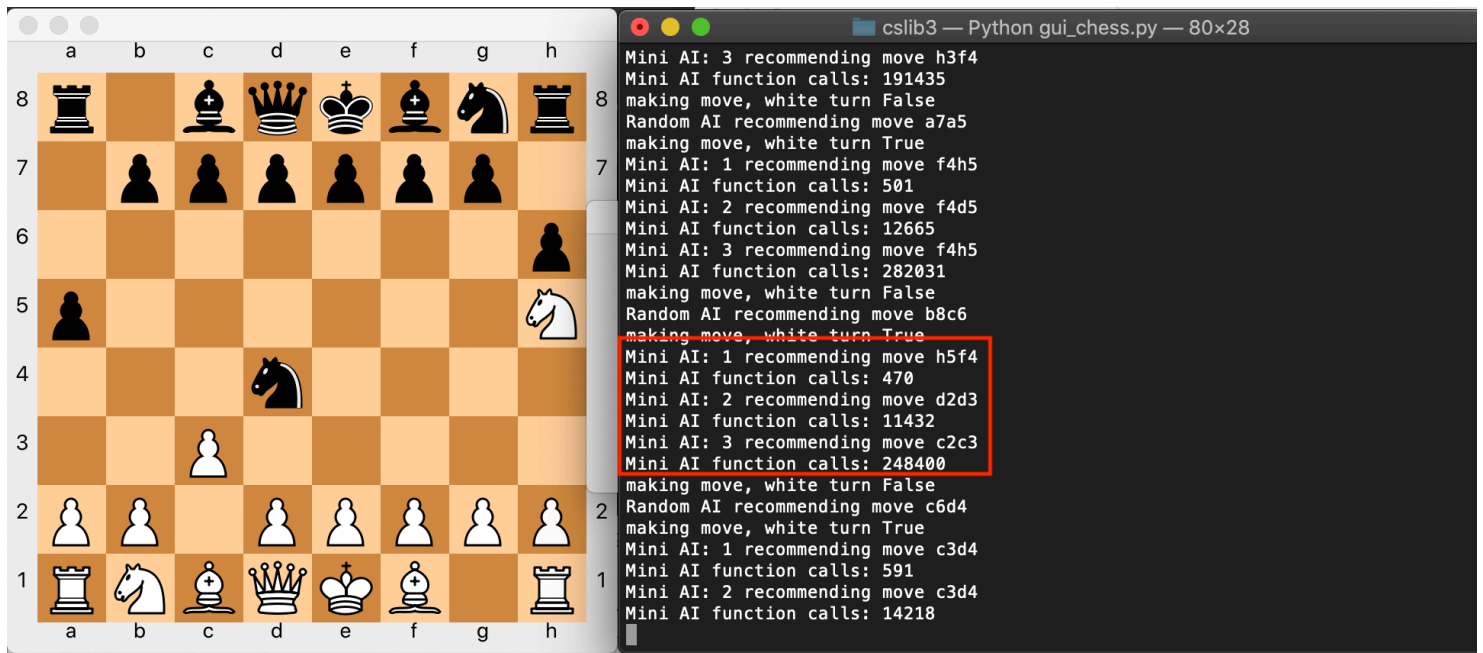
For this project, we are given python chess to build our problem state. As for displaying the actual game, Professor has provided us with the sample, which we could modify. Thus, we just need to focus on implementing algorithms.

Minimax and Cutoff Test

First, I implemented `cutoff_test`, which checks two different conditions: search reaching maximum depth, and search reaching terminal state. It returns true if any of two conditions hold and false otherwise.

Then, I wrote *Minvalue* and *Maxvalue* methods based on the pseudo-code from our textbook.

For the observation, I noticed that the number of functional calls are exponentially increasing as the depth limit increases.



Evaluation

Discussion: For the evaluation function, I gave material value to pieces. For pawn, I gave 1. For bishop and knight, I gave 3. For Rook, I gave 5. For queen, 9 and, for king, I gave an arbitrarily big number. Then, for the evaluation function, I subtracted weighted sum of oppoent's pieces from ym weighted sum of pieces. Thus, having a positive large evaluation value would mean the game is more advantageous for the AI while negative number would mean more advantageous state for the oppoent.

```
def evaluation(self, board):
    value = 0
    if board.is_checkmate():
        if board.turn == self.turn:
            value -= self.pieces_weight[6]
        else:
            value += self.pieces_weight[6]
    for i in range(64):
        board_piece = board.piece_at(i)
        if board_piece != None:
            if board_piece.color == self.turn :
                value += self.pieces_weight[board_piece.piece_type]
            else:
                value -= self.pieces_weight[board_piece.piece_type]
    return value
```

IDS for Minimax

Since there are time limits for the actual chess game, I used IDS to find different possible solutions and return the best move. I basically used Minimax search and changed the depth for each function call.

Discussion: I noticed that the optimal choice did change as the depth was deeper. However, the number of minimax function calls rose exponentially as IDS increased the depth limit. Also, given same amount of time, I noticed that IDS Minimax and Minimax in general cannot go deeper in compared to Alpha Beta Pruning.



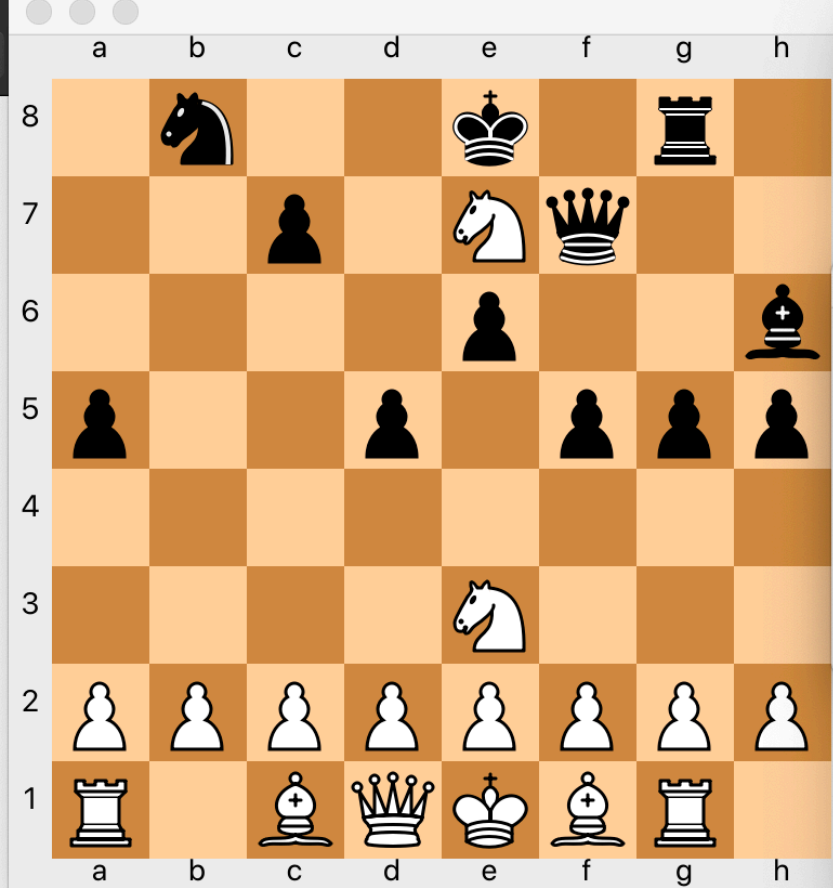
```
def choose_move_ids(self, board, depth):
    best_move = None
    init_depth = self.depth
    for i in range(depth-1):
        self.depth = i+1
        best_move = self.choose_move(board)
    self.depth = init_depth
    return best_move
```

Alpha Beta Pruning

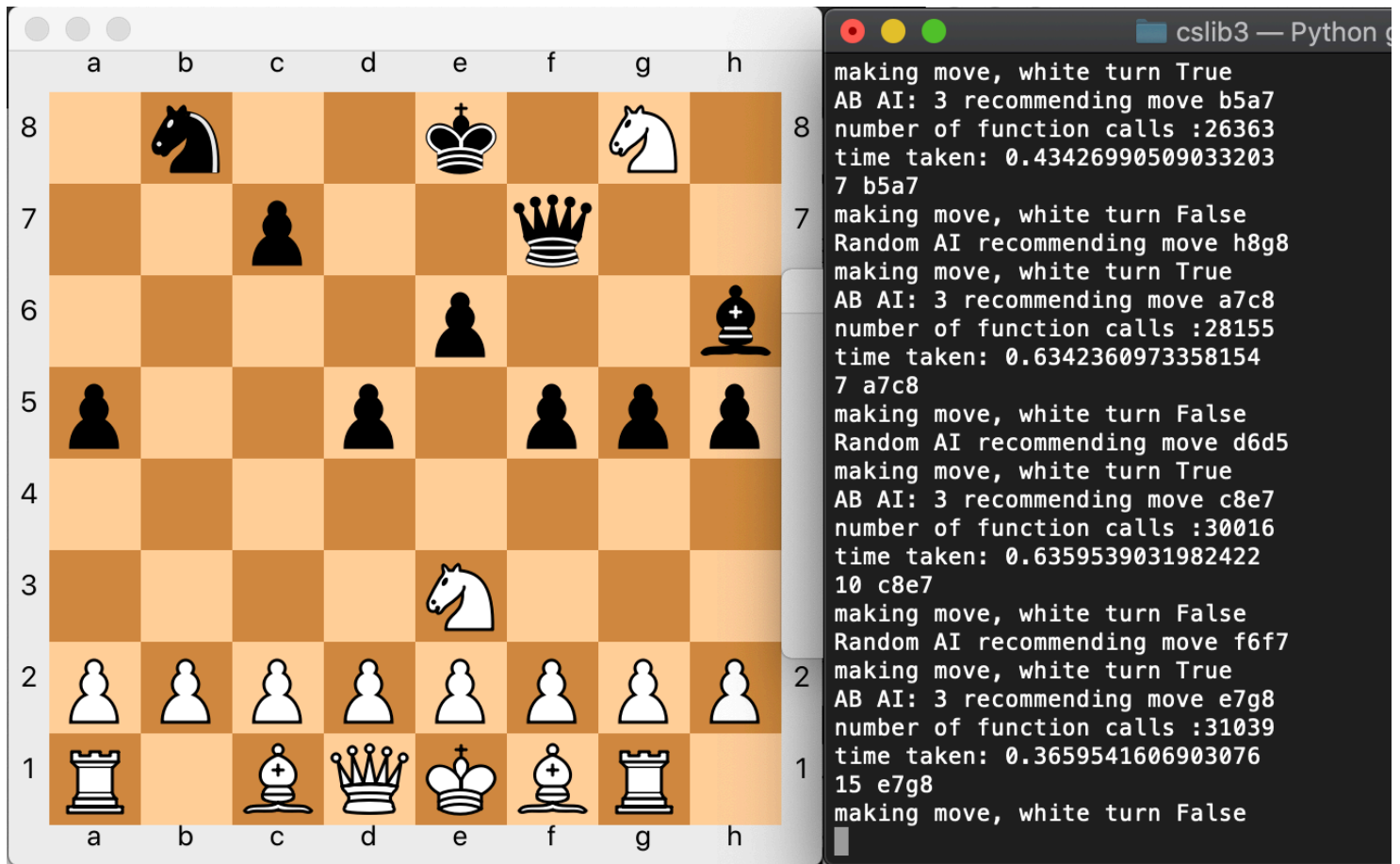
For alpha beta pruning, I used the pseudocode from the textbook and implemented the algorithm. After completing implementation, I compared two different searches. Given depth of 3, I checked to see if MiniMaxAI provides the same move as AlphaBeta.

Test: After completing implementation, I compared two different searches. Given depth of 3, I checked to see if MiniMaxAI provides the same move as AlphaBeta. To test it, I had both AIs play against the "random" player

(gives random but same moves for both players) and see if they reach same state of the board. The first one is with Minimax and the second one is with Alpha Pruning. We see that they both reached same state of the board at certain time although alpha pruning reached the state much faster.



```
cslib3 — Python
Mini AI: 2 recommending move b5a7
Mini AI function calls: 25936
making move, white turn False
Random AI recommending move h8g8
making move, white turn True
Mini AI: 0 recommending move a7c8
Mini AI function calls: 23
Mini AI: 1 recommending move a7c8
Mini AI function calls: 849
Mini AI: 2 recommending move a7c8
Mini AI function calls: 21133
making move, white turn False
Random AI recommending move d6d5
making move, white turn True
Mini AI: 0 recommending move c8e7
Mini AI function calls: 24
Mini AI: 1 recommending move c8e7
Mini AI function calls: 791
Mini AI: 2 recommending move c8e7
Mini AI function calls: 20087
making move, white turn False
Random AI recommending move f6f7
making move, white turn True
Mini AI: 0 recommending move e7g8
Mini AI function calls: 26
Mini AI: 1 recommending move e7g8
Mini AI function calls: 724
```



AB made an extra move with rook because I could not screenshot at the right time! (Thats how fast the AI was compared to Minimax)

Transposition Table

For this section, I implemented transposition table by using dictionary. Whenever cutoff test is true, the algorithm finds the evaluation value for the given board state. To avoid repetition of evaluating same boards, I used dictionary and saved the board's value. Since dictionary requires hashable object, I had to wrap a board state with a class and override hash function. I used overrode hash method by returning `hash(str(board))` instead.

Test: From the starting chess state, I tried alpha pruning with depth 6 with transposition table and without transposition table. Then, I was able to observe how transposition table can save a big chunk of time if the tree is big. Picture of the result is shown below.

```
[Seungjae-Macbook-Pro:cslib3 SJLEE$ python3 gui_chess.py
making move, white turn True
AB AI: 6 recommending move g1h3
number of function calls :451271
time taken: 166.48949599266052
making move, white turn False
Random AI recommending move h7h6
making move, white turn True
^Z
[10]+  Stopped                  python3 gui_chess.py
[Seungjae-Macbook-Pro:cslib3 SJLEE$ python3 gui_chess.py
making move, white turn True
AB AI: 6 recommending move g1h3
number of function calls :451271
time taken: 116.97733402252197
making move, white turn False
Random AI recommending move h7h6
making move, white turn True
^Z
[11]+  Stopped                  python3 gui_chess.py
```

Extension

Extension 1: For the extension, I did profiling to improve my search. By timing different functions, I figured that my evaluation is time consuming. Thus, I searched python chess documentation to find reduce evaluation method time. Then, I noticed that `chess.board.piece_map()` provides dictionary of locations and pieces. Thus, instead of iterating over the whole board to count pieces, counting over the dictionary was more efficient. The image below shows the result of two different evaluation functions. First one is the result of old evaluation and the second one is the result of new evaluation function. Old version is commented out.

```
[Seungjae-Macbook-Pro:cslib3 SJLEE$ python3 gui_chess.py
making move, white turn True
AB AI: 5 recommending move g2g3
number of function calls :22463
time taken: 6.484919786453247
making move, white turn False
Random AI recommending move h7h6
making move, white turn True
^Z
[12]+  Stopped                  python3 gui_chess.py
[Seungjae-Macbook-Pro:cslib3 SJLEE$ python3 gui_chess.py
making move, white turn True
AB AI: 5 recommending move g2g3
number of function calls :22463
time taken: 7.9567790031433105
making move, white turn False
Random AI recommending move h7h6
making move, white turn True
```

Extension 2: For the second extension, I tried to improve the heuristic function. One of the problems for the material heuristic is that it generalizes too much and often does not provide good comparison especially if none of the pieces are captured. To improve the heuristic, I used a chart from online that adds value for pieces depending on the location of pieces. The chart is shown below. AI with the chart and without the chart both played the game but the chart one did reach checkmate faster although some moves were strange. In conclusion, although this is better, it is still not the best heuristic.



```
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -2.0, -3.0, -3.0, -4.0, -4.0, -3.0, -3.0, -2.0],
[ -1.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -1.0],
[  2.0,  2.0,  0.0,  0.0,  0.0,  0.0,  2.0,  2.0 ],
[  2.0,  3.0,  1.0,  0.0,  0.0,  1.0,  3.0,  2.0 ]
```



```
[ -2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0],
[ -1.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -1.0],
[ -1.0,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -1.0],
[ -0.5,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -0.5],
[  0.0,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -0.5],
[ -1.0,  0.5,  0.5,  0.5,  0.5,  0.5,  0.0, -1.0],
[ -1.0,  0.0,  0.5,  0.0,  0.0,  0.0,  0.0, -1.0],
[ -2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0]
```



```
[  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0],
[  0.5,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[  0.0,  0.0,  0.0,  0.5,  0.5,  0.0,  0.0,  0.0]
```



```
[ -2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0],
[ -1.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -1.0],
[ -1.0,  0.0,  0.5,  1.0,  1.0,  0.5,  0.0, -1.0],
[ -1.0,  0.5,  0.5,  1.0,  1.0,  0.5,  0.5, -1.0],
[ -1.0,  0.0,  1.0,  1.0,  1.0,  1.0,  0.0, -1.0],
[ -1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0, -1.0],
[ -1.0,  0.5,  0.0,  0.0,  0.0,  0.0,  0.5, -1.0],
[ -2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0]
```



```
[-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0],
[-4.0, -2.0,  0.0,  0.0,  0.0,  0.0, -2.0, -4.0],
[-3.0,  0.0,  1.0,  1.5,  1.5,  1.0,  0.0, -3.0],
[-3.0,  0.5,  1.5,  2.0,  2.0,  1.5,  0.5, -3.0],
[-3.0,  0.0,  1.5,  2.0,  2.0,  1.5,  0.0, -3.0],
[-3.0,  0.5,  1.0,  1.5,  1.5,  1.0,  0.5, -3.0],
[-4.0, -2.0,  0.0,  0.5,  0.5,  0.0, -2.0, -4.0],
[-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0]
```



```
[0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0],
[5.0,  5.0,  5.0,  5.0,  5.0,  5.0,  5.0,  5.0],
[1.0,  1.0,  2.0,  3.0,  3.0,  2.0,  1.0,  1.0],
[0.5,  0.5,  1.0,  2.5,  2.5,  1.0,  0.5,  0.5],
[0.0,  0.0,  0.0,  2.0,  2.0,  0.0,  0.0,  0.0],
[0.5, -0.5, -1.0,  0.0,  0.0, -1.0, -0.5,  0.5],
[0.5,  1.0,  1.0, -2.0, -2.0,  1.0,  1.0,  0.5],
[0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0]
```

In the video new_heuristic, randomAi plays against AlphaBetaAI with depth of 4. The first one uses improved heuristic and wins the game in a minute while the second one that uses material heuristic wins the game in two minutes. In the video, we can clear see that second heuristic makes more threatening moves instead of making repetitive wasteful moves.

Extension 3: For this extension, I improved heuristic so that my search would find checkmate faster.

Previously, whenever checkmate occurs, I gave +/- 1000000 to the evaluation value. However, I noticed that the search algorithm does not necessary choose the fastest checkmate with my improved heuristic. Since my new

heuristic evaluates location as well, I noticed that such factor distracts the algorithm from finding the fastest checkmate. Thus, instead, I give $1000000/\text{current_depth}$ so that my algorithm would find the fastest checkmate. Demonstration is shown in a video extension³.

Extension 4: For this extension, I used advanced sorting for IDS. For any board states with evaluated values, I placed them at the front or back (sorted) depending on we are looking for max or min. However, it did not reduce down the time but increased but a lot when I tried with the starting state of the game. So, I think using advanced sorting is really inefficient when there aren't "active capture moves" in the board but moving pieces without much changes in value. Thus, sorting would still consume time while it is not really sorting. So, for further study, I think it would be nice to find when to use advanced sorting vs when to not use it. method: *advancedmovereordering*

Not Sorted

```
[Seungjae-Macbook-Pro:cslib3 SJLEE$ python3 gui_chess.py
making move, white turn True
AB AI: 1 recommending move g1f3
number of function calls :21
time taken: 0.015418052673339844
AB AI: 2 recommending move g1f3
number of function calls :84
time taken: 0.03951406478881836
AB AI: 3 recommending move g1f3
number of function calls :776
time taken: 0.4243149757385254
AB AI: 4 recommending move g1f3
number of function calls :2933
time taken: 1.0077979564666748
AB AI: 5 recommending move g1f3
number of function calls :36725
time taken: 13.64998483657837
AB AI: 6 recommending move g1f3
number of function calls :121573
time taken: 41.03745794296265
```

Sorted

```
Seungjae-Macbook-Pro:cslib3 SJLEE$ python3 gui_chess.py
making move, white turn True
AB AI: 1 recommending move g1f3
number of function calls :21
time taken: 0.016257047653198242
AB AI: 2 recommending move g1f3
number of function calls :84
time taken: 0.0856928825378418
AB AI: 3 recommending move g1f3
number of function calls :776
time taken: 0.568209171295166
AB AI: 4 recommending move g1f3
number of function calls :2933
time taken: 2.994990825653076
AB AI: 5 recommending move g1f3
number of function calls :36725
time taken: 23.970577239990234
AB AI: 6 recommending move g1f3
number of function calls :121573
time taken: 120.55181908607483
```

Extension 5: For this extension, I made a very simple open book that uses "pawns" first. The picture is shown below as well as video demonstration. To use the book, set self.use-open-book to True in AlphaBeta object.

