report.md

# Chess

Assignment 3
CS 276, Artificial Intelligence
Fall 2021, Dartmouth College
Spencer Bertsch

This report contains three sections, as seen below:

- Description
- Evaluation
- Responses to Discussion Questions

# Description

1. How do your implemented algorithms work?

### Minimax

The minimax algorithm works by taking some current state of a graph (in this case a board object) and iterating over all the possible moves that one of the players could make. For each iteration, it calls *min_value()*, one of the two recursive parts of the Minimax algorithm. For each move, the *min_value()* function makes recursive calls to *max_value()* which calls *min_value()*, etc. until the game is complete, or a depth limit has been reached. Once a base case is reached, a utility value *u* gets passed back up through the recursive calls to the outer function, and that *u* value is then associated with the initial, given move.

This is how vanilla minimax works! The time complexity of MiniMax is O(b^m) where b is the branching factor (number of legal moves in our case), and m is the max depth of the tree, or in our depth limited search, the MAX_DEPTH parameter. The space complexity is only O(b*m) because as each space is searched, we don't need to keep all old nodes in memory because that part of the game has already been played and we no longer need the information stored in that part of the tree. As we can see, the time complexity is clearly the limiting constraint for the MiniMax algorithm. We can see this empirically later on when time tests are run for different MAX_DEPTH values for depth-limited minimax.

### Depth-Limited Minimax

Same as above, but we just make sure to add the MAX_DEPTH to the minimax so that we don't have to search the entire tree. For a game such as chess where b=35 and m~=100, so the time needed to explore the entire space would be incredibly large, making the problem intractable. In order to solve this problem, we used depth-limited Minimax, an algorithm that has an additional check in the recursive function that checks whether the depth of the tree is greater than some MAX_DEPTH threshold. If it is, then it only explores the tree up until that threshold is reached and no further. Implementing depth-limited Minimax is crucial because chess has such a large branching factor and such a deep tree. The performance (speed and nodes visited) for searches with MAX_DEPTH of 1, 2, and 3 are shown below.

### Alpha Beta Pruning

Alpha Beta Pruning works generally the same way as MiniMax, but with a few added constraints that gave a significant performance increase to the run time of the algorithm. The idea behind Alpha Beta pruning is that there are entire subtrees that we don't have to search as long as we know that the Minimax algorithm will never consider those nodes. In order to do the bookkeeping for these nodes, we use two ints *alpha* and *beta* that we pass along to the min() and max() functions. In addition to the original Alpha Beta Pruning algorithm seen in the textbook, move ordering has also been used to further increase the performance by reducing the number of nodes checked during search.

### Iterative Deepening Minimax

Iterative deepening minimax works the same way as minimax, but we run the algorithm for multiple depths and choose the best utility for all depths. This implementation was similar to the iterative deepening search we saw in the Chickens_and_Foxes assignment, but instead of returning the single solution and exiting when the solution is found, we return the solution and continue searching until the MAX_DEPTH has been reached. When that occurs, the best_move is then found and returned so that the move can be made.

2. What design decisions did you make?

For Minimax IDS-Search, I stored all the 'best_moves' in an outer dictionary so that I could keep track of if/how they changed. Doing this allowed me to look at how the utility function was calculating the relative utility of different moves on the same board because it had the capability to explore to different depths. I know the assignment mentioned that it would also be a good idea to create an instance variable *self.best_move*, and although this would also be a perfectly reasonable implementation, using a dictionary instead allowed me to easily keep track of how the best move changes as MAX_DEPTH changes.

# Evaluation

1. Do your implemented algorithms actually work? How well?

Depth-limited Minimax, Alpha Beta pruning (with and without move ordering), and Minimax with depth-limited search all perform exactly as expected.

We can see that with Alpha Beta pruning, the number of nodes searched and the time taken to find the best move is significantly better than the Depth-limited minimax and the iterative deepening minimax. For Alpha Beta Pruning, we can see that adding move ordering increases the performance of the algorithm by reducing the number of nodes that get visited during the search. We can also see that using iterative deepening gives us different suggested best moves from Minimax as the search space is explored with differing max depths.

### Depth-limited Minimax

Testing the performance of the different implementations of minimax and alpha beta pruning was initially a difficult task, but by examining factors such as the number of nodes visited along with doing timed performance tests, it was easy to see how well each algorithm performed.

When the MAX_DEPTH is set to 1, the Minimax player can clearly see a single turn ahead, but no more. It will trade a queen for a pawn if that looks like the best move in that position, even if that places the queen in a very vulnerable position later on. When MAX_DEPTH is set to 2, Depth-limited Minimax displays much better performance. It plays more defensively, keeping most important players back until there is a clear opportunity to take an opponent's piece without sacrificing anything.

### Alpha Beta Pruning

Alpha Beta performs the same as Minimax under identical conditions, but Alpha Beta is much faster visiting far fewer nodes and requiring much less time to run. In order to ensure the algorithm performs the same as Minimax (makes the same decisions in the same situations), I ran a series of manual tests by playing games between a human_player (me) and either Minimax or Alpha Beta. See the appendix for a few of the decisions that were made by Minimax and by Alpha Beta, showing that they made the same decisions in the same environment.

| | Move 1 | Move2 | Move 3 |
|---|---|---|---|

| | Move 1 | Move2 | Move 3 |
|---|---|---|---|
| Without Reordering | 2,005 Nodes Visited | 2,032 Nodes Visited | 4,778 Nodes Visited |
| With Reordering | 1,651 Nodes Visited | 1,497 Nodes Visited | 2,150 Nodes Visited |

We can see that after implementing a simple move reordering, the number of nodes visited was reduced by up to around 50%. Perfect node ordering should increase the performance even more to O(b^(m/2)), and with random we should achieve performance closer to O(b^(3m/4)), which is still much better than the vanilla Minimax's O(b^m). There were significant gains seen by implementing Alpha Beta pruning instead of Minimax, and implementing move ordering increased the algorithm's performance even further, as seen in the above table.

### Iterative Deepening Minimax

Iterative deepening minimax performs similarly to minimax, but finds the best_move over several maximum depths before returning the *absolute* best_move. There are many ways to implement iterative deepening, but the way that I chose to keep track of the absolute best move over multiple MAX_DEPTH values was to simply store the move and utility value for each depth in a dictionary, then return the absolute_best_move.

# Description

See the below tables for a comparison of the run times and nodes visited for each different search under the same board conditions.

### Max Depth: 1

| | Measured Time (sec) | Nodes Visited |
|---|---|---|
| Depth-Limited Minimax | 0.049 | 497 |
| Alpha Beta Pruning (no reorder) | 0.049 | 497 |
| Iterative-Deepening Minimax | 0.050 | 497 |

### Max Depth: 2

| | Measured Time (sec) | Nodes Visited |
|---|---|---|
| Depth-Limited Minimax | 0.952 | 13,081 |
| Alpha Beta Pruning (no reorder) | 0.262 | 3,270 |
| Iterative-Deepening Minimax | 1.016 | 13,578 |

### Max Depth: 3

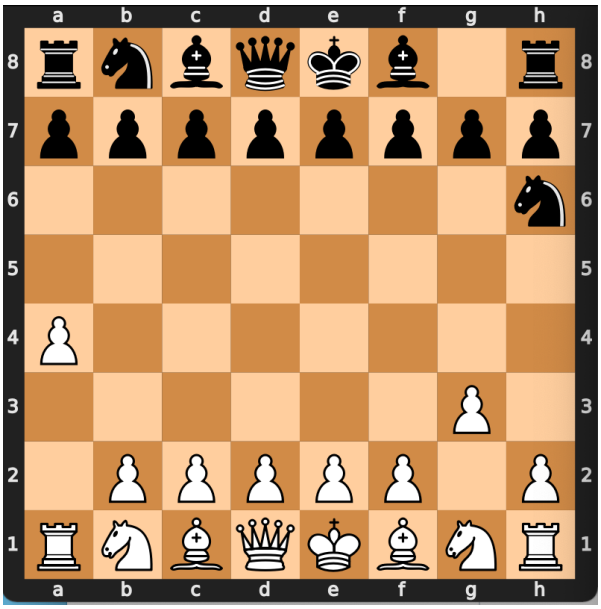
| | Measured Time (sec) | Nodes Visited |
|---|---|---|
| Depth-Limited Minimax | 16.164 | 227,601 |
| Alpha Beta Pruning (no reorder) | 1.130 | 15,077 |
| Iterative-Deepening Minimax | 16.899 | 241,179 |

1. (Minimax and Cutoff test) See above tables to see how each algorithm performed in terms of calls made to Minimax and the time taken. We can see that the number of nodes visited using Minimax and Iterative Deepening was far greater than the number of nodes visited using Alpha Beta pruning.

2. (evaluation function) The evaluation function was one of the key concepts in this assignment. The evaluation function uses a concept described in the textbook in which each piece is given a certain weight (1-Pawn, 3-Rook, 3-Knight, 3-Bishop, 9-Queen) and the board state is evaluated purely on the weighted average of white and black before and after the move in question is made. The Minimax algorithm always prefers to take higher value pieces so that the board will be in a better state after the move has taken place. As MAX_DEPTH increases, the evaluation function is called more, causing the time taken to search to increase dramatically. In order to determine whether it was the white or black player's turn, I used *board.turn* in conjunction with a new instance variable *player1* which was True if the current player is white. With that it was easy to calculate the utility for either the white or black player depending on the weighted sum of the pieces still on the board.

3. (alpha-beta) The Alpha Beta algorithm is performing as expected. As mentioned above, we know from the textbook that perfect node ordering should increase the performance even more to O(b^(m/2)), and with random we should achieve performance closer to O(b^(3m/4)). See the table in the **Evaluation** section to see a detailed outline of how move-ordering affected the number of nodes searched. We can see from that table that when we use move ordering and have a MAX_DEPTH of 2, the number of nodes searched is still decreased by around half in some cases. This is the type of behavior that we would expect to see after implementing move ordering in the Alpha Beta algorithm.

4. (iterative deepening) Iterative deepening works by finding the best move at different depths, then choosing the move that has the highest utility among them.




We can see in the above example that when depth=1, the Iterative Deepening Minimax algorithm can't see far forward enough to see any moves that have a non-zero utility. It chooses to move the rook in h8 into the g8 position. This move signifies the Minimax algorithm couldn't find a move that had a good utility, so it's just returning the first zero utility move: moving the rook slightly in a safe way.

| Iterative Deepening Depth | Move | Utility Score |
|---|---|---|
| Max Depth 1 | h8g8 | 0 |
| Max Depth 2 | h6g4 | 1.029 |

When iterative deepening continues and max depth is increased to 2, however, we can see that the utility associated with advancing the knight is 1.029 which is greater than the initial move with a max utility of 0. This better move is the one that gets returned by the Iterative Deepening Minimax algorithm.

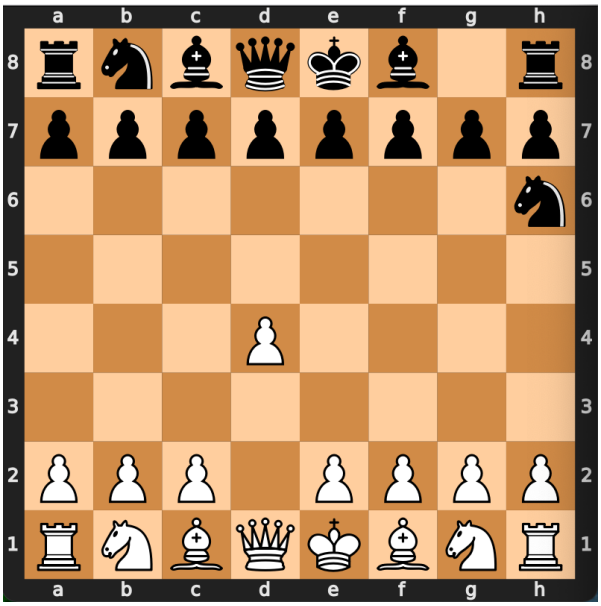
## Additional Notes

One thing that took me a while to figure out, that I initially mistook for a bug, was that the Minimax or Alpha Beta algorithm seemed to make misplays in the late game. After most of the opposing player's pieces were taken, the Minimax or Alpha Beta algorithm would miss the opportunity to take a high value piece from the opposing team and move a rook or pawn on the other side of the board instead. The reason for this behavior is that the utility function values putting the opponent in check above all else, so later in the game when the opposing player's king is exposed more often, the Minimax or Alpha Beta player would ignore all opportunities to take knights, rooks, bishops, or even queens if it meant they could put the other player in check.

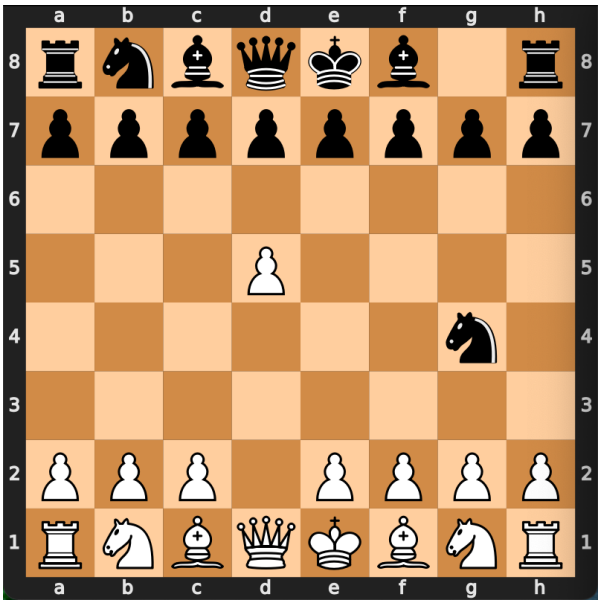
Although this was generally an acceptable way to play, it allowed the opponent to make much more aggressive plays, putting their queen in danger in order to take an important piece as long as they could protect their king from check mate. In order to counteract this, I experimentally changed the utility associated with putting the opponent in check until other plays such as taking their queen were more valuable. This strategy worked somewhat well, but each methodology has trade-offs. Ultimately a better solution would be to have a dynamic utility function that is better at representing the state of the board. The textbook describes other pitfalls of the simplistic approach that we have implemented including the horizon problem, and I can see that using an algorithm with learned parameters that are updated through repeated simulation might be better equipped to deal with difficult utility calculations.

# Appendix

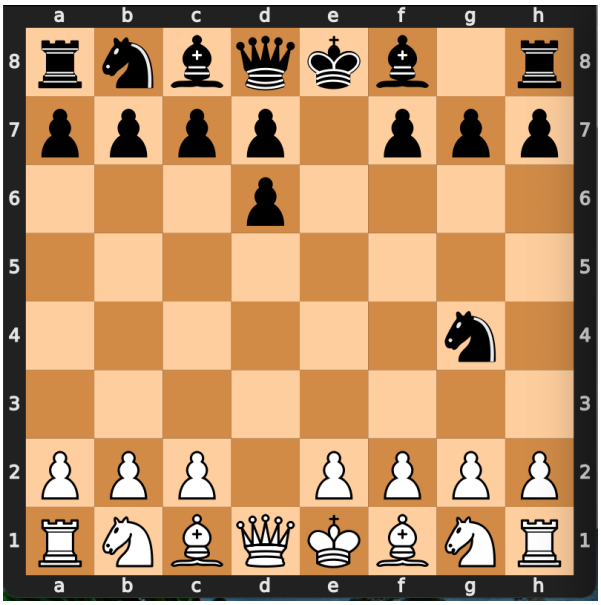
This is an example of how the Minimax and Alpha Beta algorithms make the same decision with identical board states. In this case white is a human player, and black is Minimax or Alpha Beta. For this first move (white d2d4), the Minimax or Alpha Beta chooses the first of many moves with max utility, pushing their knight into the board.




Similar to the first move, white advances the pawn and the Minimax / Alpha Beta algorithm chooses one of the many max utility moves which happens to be to continue to advance their knight.



This is where things start to get interesting, now that there is a clear move that increases the utility for black, the Minimax / Alpha Beta algorithms take the white pawn with no drawbacks.



Lastly, after a very foolish move by white leaving the queen exposed, the Minimax / Alpha Beta algorithms move in and take the queen with the bishop, largely increasing the relative utility for black.