CS5100 Final Project
December 12, 2019

Rohan Subramaniam
Rajesh Sakhamuru

Minimax Chess AI Project

Abstract

This project was building a chess game with an artificial intelligence agent to play against. First, castling and en passant maneuvers were implemented followed by game logic for check mate and tie scenarios. Once the game environment was fully operational, an AI agent was constructed using depth limited minimax search with iterative deepening, forward pruning, and alpha beta pruning optimizations. The heuristic evaluation function used in the minimax takes input from a genetic algorithm that generates weights for the evaluation metrics over several generations. The result of this project is a chess agent that actively moves towards attacking the king and capturing opposing pieces while attempting to keep its pieces safe.

Background

Chess is a widely understood and well researched game with decades of recorded attempts at creating artificial intelligence players. All of these factors lend themselves to an interesting and challenging final project that can indefinitely be expanded and improved. Furthermore, the adversarial nature allows for the use of minimax search along with the various optimization strategies we have studied in artificial intelligence. With all this in mind, we chose to build a chess AI from the ground up using our own game logic, evaluation metrics, and minimax optimizations.

We successfully built a chess agent that actively tries to progress towards a winning state while protecting its own pieces. The agent consistently beats random moves, and will also usually beat an inexperienced player who fails to consider certain possibilities in the board state. Even while playing against a higher quality opponent, the agent makes mostly reasonable moves and is not a complete pushover that can be beat without some real thought. Our agent won't be winning any awards, but it is somewhat intelligent in its chess play. While building our AI was not without its challenges, we built a respectable chess AI that can continually be adjusted and improved moving forward.

Outside Resources

Implementation of AI into this project involved the use of some outside resources to create the User Interface using python3's built-in Turtle Graphics Library. Using a pre-built chess UI, found at https://repl.it/@f9we/chess by user @f9we. @f9we's original code was in python2, so the modifications needed to use the code in python3 were made and the chess UI was borrowed from there. The borrowed code also included verifying that each move was possible before moving it on the board, and also had many bugs in that verification process (which mostly involved false-positives, allowing many moves that are illegal).

AI Concepts

We used several adversarial search AI concepts to build our AI player around minimax.

1. **Minimax algorithm**

   The core decision making process behind the player is a minimax algorithm that we optimized. We implemented a minimax search that accepts a variable heuristic that is generated from our genetic algorithm. The algorithm first generates the list of possible moves for each side and the list of threatened spaces by each side to be used in the heuristic evaluation function. Minimax then generates all possible children states from the list of possible moves by deep copying the board and making the move. Terminal states return the heuristic value of the board and non-terminal states recursively call minimax until a terminal state is reached and its heuristic is returned back up. The minimax algorithm alternates between maximizer and minimizer's turns and chooses the child state with the highest and lowest heuristic respectively. Each recursive call passes the opposite of the current player to ensure minimax alternates based on whose turn it is. In this version of minimax search, the maximizer is always the black side AI player and the maximizer is the white side human player.

2. **Depth-Limited Minimax and Alpha-Beta Pruning** (and board heuristic)

   The minimax algorithm without a limited depth search would have to play out all possible games to a win/loss/draw result from a given board position. Depth-Limited minimax stops our minimax search up to a given depth (in our case 4) with terminal nodes that do not necessarily represent a guaranteed win/loss/draw scenario.

   In order to evaluate all these states, a heuristic was developed which took into account:

   - The positions on the board
   - The distance of each piece (and threatened square) to the opposing king
   - The pieces on the board threatened by each side of the board
   - Encouraging moves that threaten center rows and columns

   The values in the heuristic that determine the weightage of each of these positional instances are optimised via the Genetic Algorithm (as explained at #5). The heuristic function (evaluate(board_state, pieces) in class opponent_AI) is optimized from the perspective of the black side of the board, although the code could be modified to use minimax for the white side of the board. The evaluate() function returns either a value of:

   - -1000000 for a white victory
   - +1000000 for a black victory
   - 0 for a draw k
   - 999999 to -999999 for any other board state

   Board states that favor black side are more positive and if a state favors white side it is more negative. In this implementation only black uses the AI black moves are always maximized and white are minimized when predicting the best moves for each player using minimax.

Alpha-Beta pruning keeps track of the best value for both the maximizer (black) and the minimizer (white). Using this value, if the minimax search-tree encounters a branch that is guaranteed to have a worse result than the best currently guaranteed, then it is not searched, or "pruned." "Alpha" keeps track of the best move for the maximizer, and "Beta" keeps track of the best move for the minimizer.

The result of this "pruning" is a much smaller search space, subsequently decreasing memory usage, while still having the exact same result. The time for the minimax algorithm to choose the best move for black, due to the smaller search space, also decreases significantly. Prior to implementing alpha-beta pruning, performance was significantly worse, with a depth of 4 search taking about 30% longer.

3. Iterative deepening

Depth 4 minimax is most often calculated in a couple of minutes, but there are some cases with a large search tree where the depth 4 minimax search takes considerably longer. In order to avoid this we implemented iterative deepening to first run the minimax search with a depth of 2, which takes seconds compared to minutes, and subsequently run the depth 3 and 4 search. We have a 4 minute time limit set while the minimax runs, which forces the loop to break and return a non-answer if the depth 4 search does not finish in the alloted time. In the case that the search was not completed, the result from the depth 3 search is returned instead.

Although most iterations the depth 4 completes within the time limit, we observed a few outliers where it took several minutes more. Instead of waiting for an indeterminate amount of time, we chose to enforce the time limit and use the best guess possible within the "thinking" time alloted to the AI.

4. Forward pruning

In order to improve runtime efficiency while evaluating the search tree, we implemented forward pruning before expanding each level of search. Since a new board and evaluation is created for each possible move from the current state, excluding some of the more unfavorable moves from being expanded upon cuts down on the size of the search tree. To address this, we implemented a check at the beginning of each minimax iteration. Any time there are greater than 20 possible moves from a given state, the children states are evaluated as is and sorted. The bottom 50% of children states are discarded, and the top half get evaluated to continue the minimax search.

The addition of forward pruning actually slowed the minimax search down for depths 1 and 2 because it calls for evaluating each child state even in the non-terminal nodes. However for the more expansive depth 4 search, the efficiency gained from pruning half the nodes from each level of the search tree far outweighed the time lost to evaluating each state, resulting in an overall speed increase.

The main drawback of forward pruning is the possibility of missing a future optimal move subsequent to one of the excluded "suboptimal" moves. To make sure that there is sufficient balance between efficiency and optimality, we set the pruning threshold to 20 moves, so every state has enough variety to choose from. Most game states average around 20-40 possible moves, but some can have many more valid moves, in which case forward pruning reduces the search space significantly.

5. **Genetic Algorithm**

A genetic algorithm is used to come up with optimal weights for the heuristic metrics which are used to grade chess board states for the minimax algorithm.

The population size of the potential lists of heuristic weights is 50. Each list of heuristic weights in the population is graded using the fitness() method. There, they are tested against a variety of static board states to determine an acceptable heuristic that can grade board states reasonably well. The goal score is ‘0’ and the grade/score is higher based on how many states/conditions the list fails to grade correctly.

Each parent population after being graded is “evolved” by retaining a portion of the best graded lists from the parent population to the children population. A few more lists are randomly selected for the children generation and then they all have a low chance of being given random mutations.

Then, the remaining lists of the children generation (to have the same population size as the parent generation) are produced by randomly combining lists that have been selected so far. This “crossing-over” creates hybrids of the chosen lists, increasing biodiversity and the chance that the score can be improved as the population gradually decreases the average score until a list gets a score of ‘0’.

The more generations it takes to get to a list with a score of “0,” the more randomness is increased up to a certain point in order to better get lists out of local minima through random chance. If a number is randomly changed and decreases it’s score, then that number will be more likely to propagate throughout the population in future generations getting closer to the goal. If there is no list that scores ‘0’ even after 50 generations, then the lowest scoring list from the final generation is returned as the solution.

Even after using the genetic algorithm to optimize the heuristic function, a few things were changed manually like increasing the value of the queen on the board which wasn’t accounted for as accurately as it could have by the test-boards that were used in the fitness function.

Future work

One issue with the game logic we encountered involves the checkmate checks for the human player. The game logic prevents the AI player from making any illegal moves, but does not prevent the human player from moving into check or checkmate. Since the human player can more easily distinguish these scenarios, we focused on ensuring the AI could not make any illegal or game-breaking moves and will revisit the issue with the human player at a later time.

The complexity of chess and the vast search space due to the huge number of potential scenarios mean that our chess AI can always be improved. The heuristic evaluation function we used already accounts for pieces on the board, piece mobility, pieces threatened, and primitive center board control, but there are several other factors that can be considered to improve accuracy.

To improve on the current version, we can use the genetic algorithm. The genetic algorithm attempts to optimize the current heuristic by providing weight to specific aspects of the evaluation function. By adding test-board scenarios to the fitness() function, the heuristic can be further fine-tuned to optimally evaluate the chess board.

We can also modify the evaluation to also consider the structure of the pawns overall, trapped pieces, tempo (move efficiency), game phase (early, mid, late), and connectivity. Each of these additional metrics for the evaluation function requires additional calculations for each call of the evaluation function and will subsequently increase runtime in exchange for a more accurate heuristic.

One way to improve runtime in general would be to restructure the chess board and pieces to be more efficient with generating children and creating a search tree. The skeleton code we used for the board is inherently quite slow because it involves turtle graphics in addition to creating a new deep copy of the board *and* pieces objects for each iteration of minimax. Ideally we would like to be able to run minimax on just the array of pieces representing the board and avoid copying all of the other fields that go into the board and piece objects. This would reduce the cost of generating children considerably and therefore speed up the minimax search because of the high frequency of the generate children call. Another way to approach this issue could be to process the board states separately from turtle and then feed the chosen move to a graphical engine for the chess board. This method would disconnect some of the components of the graphics from the chess game itself and decrease runtime.