Implementation of a Chess Playing Artificial Intelligence

Sean Stanek

vulture@cs.iastate.edu
Overview

Chess has always been a hobby of mine, and so have computers. In general, I like creating computer programs that can do things that humans could never dream to do, and as such, a computer artificial intelligence that plays chess is just such a program.

Goals

For this project, I plan to implement a working chess AI. The goal is to see how it responds to tweaking various variables, such as how offensive or how defensive it plays, point values of different pieces on the chessboard, branching factor, and search depth. I assume that an exhaustive search algorithm would be exponential in order, so a large depth might be unfeasible. However, a worthy goal of this project would be to create a chess AI that could exceed my own chess playing abilities.

Implementation

The obvious solution to finding the best move to make next on a chessboard is to exhaustively search the entire search space of any given state on a chessboard. There are 16 chess pieces on each player’s side, each of which moves and captures enemy pieces differently. Additionally, to search the entire search space, each of these pieces should be moved to any and all spaces it can be. I score should be assigned to that move, and the best score should be used to decide which move to make.
This seems easy enough, but then realize that a chess AI that can only look at the current state and not into future states is no AI at all. Player A could be one move away from certain checkmate of Player B, but Player B would not even see this if he did not look even one move into the future.

So, the obvious solution is to look at every possible combination of countermoves for every combination of moves, and try to figure out what the opponent would do. In fact, why not check every possible combination of countermoves for every possible combination of every possible combination of every possible…

Not possible. There are infinitely many combinations of moves much less sequences of moves that one can make on a chessboard. So, obviously a line must be drawn somewhere. Of course, the main obstacles that would be approached are space and time. Obviously using a depth-first search would not be too taxing on space, but definitely time. A breadth-first search would be taxing on both space and time. So, a depth-first search would be simplest and most efficient for now.

For each move that the computer player can make, a recursion should be made. Each new chessboard’s state should have all of its moves checked and scored. The best scores should be returned, and ultimately, the best score used in the decision to make a move. Because the AI will need to predict what the human player (or other player in general) will do, depending on the current state’s player’s turn, pieces captured by the AI will be
added to the score, and pieces of the AI’s that get captured will be subtracted from the score. This is simply recursed to a desired depth, based on available time.

Toward the middle of a game, as both players open up their sides of the chess board, more and more pieces will be exposed and have more opportunities to make different moves. At the beginning of a chess game, a player can make only one of twenty different possible moves. In the middle of a game, a player could easily have sixty different possible moves. This could be a lot more taxing on time constraints, so instead of allowing all possible moves, it might be better to limit the search space by pruning the nodes with the current lowest scores before continuing on to recurse further. Obviously, if your king were about to be captured, you would want to eliminate that move from even being a possibility. So, simply eliminating that move from the current search space will prevent that path of losing from being taken. Modifying one or both of the maximum branching factor and maximum depth of recursion could change how the AI can work in a given time constraint.

Obviously since we are not doing a completely exhaustive search of the game, we have to assign power values to each of the chess pieces to aid in deciding how to play the game. Each chess piece is assigned a value – pawns are worth one point, knights and bishops are worth three points, a rook is worth five points, a queen nine points, and a king 200 points (you would never want your king to be captured). These numbers are just generally accepted numbers; however, there is no guarantee that a computer could actually have a better game with these numbers. If it placed more value on keeping its
knights instead of the rooks, it might in fact have a better chance of winning. This is one set of values that can be tweaked.

Another set of values to be tweaked is the likelihood that the AI will attack or defend. Should capturing a rook be as important as keeping one’s own rook? This can be different depending on the opponent to be played. If the opponent is a strong attacker, a good defense will frustrate the opponent (it certainly did to me). So, a couple values to assign more points for capturing a piece (attack) instead of losing a piece (defense) can be used to decide this.

**Analysis**

At any given state, there could be usually 20-60 possible moves for a side. Let's call this number K. This number could be limited by a beam search maximum branching number. By trial, 20 seems to be a reasonable number that doesn't prune too much, but isn't too slow to go through many depths (the game state gets more complex as more pieces open up - a queen in the middle of the board could make 20 different moves by itself easily). We will call the pruned maximum branching factor k. The algorithm used is an exhaustive depth-first search that will terminate at a given depth d, and then continue throughout the rest of the search space. Additionally, the first level is always fully searched (i.e. not limited to the beam-search limit). Assuming this first level is depth 0, this means we have to perform $O(K^d)$ node visits. In each node visit, we sort the array of scores in order to choose the best score, but also to use as the top N scores in the beam search. This sorting algorithm used is a bubble sort, but could actually be done with a
qsort or something better. But, being implemented as a bubble sort (there's only around 40 items to sort anyway), this equates to a runtime of $O((K^d)K^3) = O(K^3k^d)$. In implementation, I can search through about 1.5 million nodes per second. Running a beam search, we can run a depth of four in a couple seconds. Of course, to get the most powerful computer AI, pruning nodes may not be an option, especially since many moves in chess are unseen combinations of other moves. This could require that no nodes be pruned, and a large depth be used. Assuming an average of 30 moves per state, and a maximum depth of 10, this would require visiting $30*30^{10}$ or 18 million billion nodes, taking roughly 12 billion seconds or 374 years. It is obvious why chess is such a complex problem, and why computers have only recently been able to consistently defeat chess grandmasters, and running in massively parallel configurations at that. It is obvious that pruning MUST be done, and even then, depth is really a limitation. In practice, playing against my own AI, I enjoy using full searches (no beam width limit) and a depth of four. This takes the computer anywhere from five to twenty seconds (which is acceptable), and it is very intelligent and can predict many of my attacks before I make them. With a more specialized pruning algorithm, it should be possible to prune out more redundant nodes and be able to extend this AI beyond a depth of four, making it much more difficult to defeat.

**Conclusions**

A “perfect” chess AI is almost completely impossible. As opposed to simpler games, chess simply has a vast search space, both in the amount of possible moves that can be performed in each state and also in the number of moves needed to finish the game.
Searching all moves at every given state is time consuming and spends more time searching nodes that might not have to be searched, and does not allow the AI to search very far into the future of the game to predict what its opponent will do next, and thusly what move it should make next. On the other hand, pruning nodes and especially pruning them early on can allow the AI to search very far into the future, but at the cost of an incomplete search (and possibly losing the game long before it knows it could even possibly lose the game). The chess AI that beat world champion chess player Gary Kasperov was powered by one of the largest supercomputers that IBM could afford to throw at the task, as well as having an additional set of training data of openings and strategies that have defeated Kasperov in the past. Certainly powerful computers can beat the average player, but for how much longer can powerful chess masters beat the average computer?