Kevin Rhoades

May 11, 2020

CS730

Chess AI Using Minimax and Alpha-beta Pruning

Abstract

Chess is a game with a large state space. For this reason, creating an AI that can play reasonably well while also not taking too long to find a good move is an interesting problem. The chess AI was made using minimax and alpha-beta. The alpha-beta algorithm was much faster than minimax at all depths.

The basic evaluation function used for minimax was based on the piece values left on the board. This evaluation function worked well for later stages of the game, but it created many “stale” games where each AI made the same moves over and over if the game was in an early stage of the game. Another evaluation function was made which accounted for the amount of legal moves each piece could make. The best weight for this evaluation function to be weighted with the piece value was 0.35. This new evaluation function decreased the number of ties along with increasing the win rate.

Problem Statement

Chess is a full information, turn based game where the objective is to kill the enemy’s king. Other pieces on the board are used to achieve this goal. In my project I looked into making a computer play this game. Chess has a very large state space, and an average branching factor of 35(Stuart Russell & Peter Norvig, 2010, p.162). This means that if we only want to explore to a depth of 6 moves, we have to explore ~2 billion states. If we want to explore a depth of 7 moves, that number jumps to 78 billion states, which would take about 9 days to explore even if we are exploring 100,000 nodes per second. Since the average chess game lasts a lot more than 7 moves, it isn’t feasible to search the whole state space of chess.

Because of this large state space and large branching factor, having AI play chess is an interesting problem to solve and narrow down this state space to a manageable amount of states to explore, while also having the AI play at a reasonable level.

The Solution to the Problem

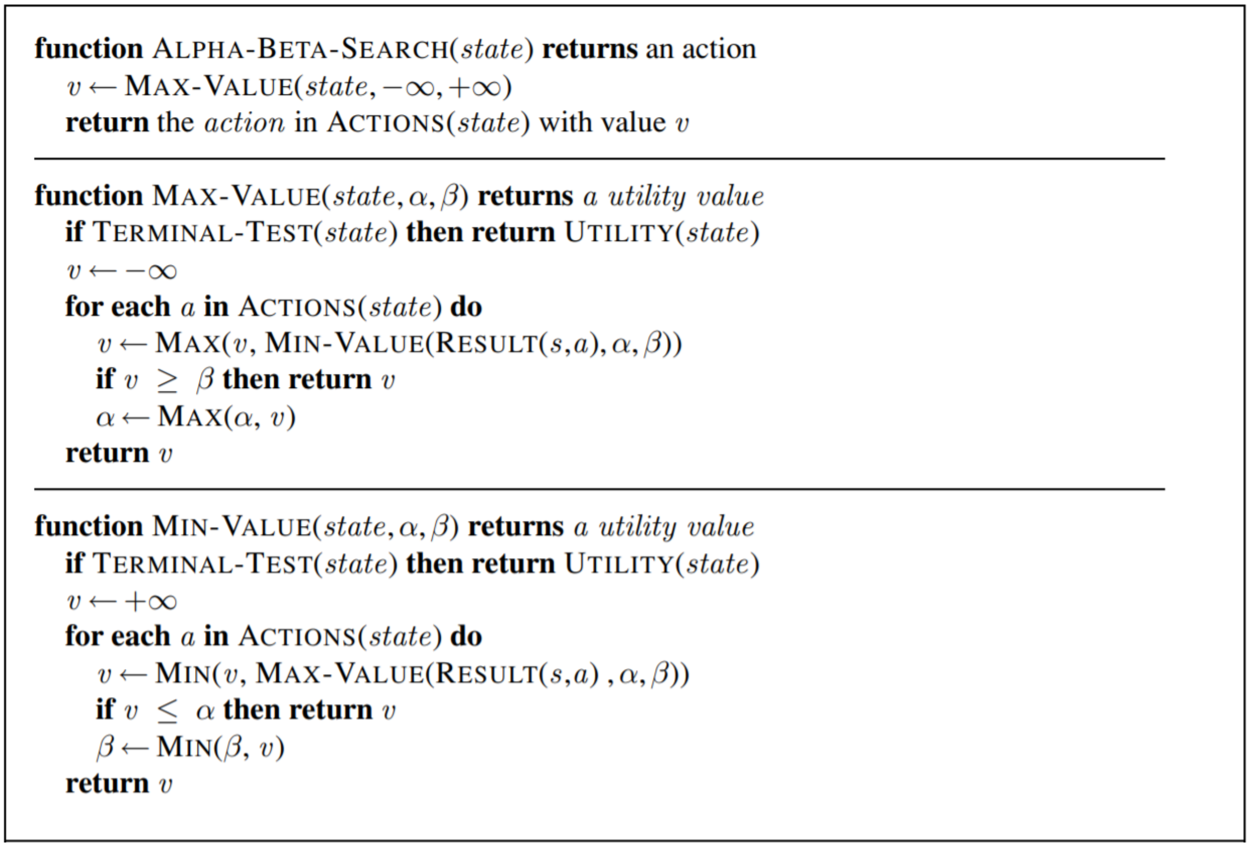
Since there are two players in chess, and each have conflicting interests, we can’t search through the state space using a regular search such as A*. Instead, there is a player looking to maximize a score in the game, and the opposite player is trying to minimize that score. That score in this algorithm for chess is the evaluation function. At each depth, the nodes switch between maximizing the score and minimizing it because each player gets 1 move each turn in chess. At each max node, the node chooses the highest score possible out of its children, and at the min nodes it does the opposite.

```
function MINIMAX-DECISION(state) returns an action
  return $\arg\max_{a \in \text{ACTIONS}(s)}$ MIN-VALUE(RESULT(state, a))
--------------------------------------------------------------
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow -\infty$
  for each a in ACTIONS(state) do
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s, a)))
  return v
--------------------------------------------------------------
function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow \infty$
  for each a in ACTIONS(state) do
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

Minimax algorithm (Stuart Russell & Peter Norvig, 2010, p.166)

If this was a game with a small state space (like tic tac toe), we could simply call the minimax function at each state and know the exact outcome of the game. Unfortunately for chess, the state space is large so we have to depth limit the search. Instead of returning a win or a loss at the depth limited nodes, we return the evaluation function. This function should be a good indicator as to which side has the advantage in the game. For chess, my basic function (I call it "material" evaluation function), is (value of white's pieces on the board) - (value of black's pieces on the board). The values of the pieces are: Queen worth 10, Rook worth 5, Bishop 3, Knight 3, Pawn 1, King 1000. In this case, white is maximizing, and black is minimizing.

With just minimax, my chess AI would be pretty bad or slow. This is because it's exploring a lot of nodes it doesn't need to explore. For example, if a min node has pulled up a value of 5 from one of its children, and its direct child max node gets a 6 from one of its children then the algorithm can "prune" off any other children of the max node that found the 6. This is because the max node will always choose something greater than or equal to 6, but the min node above it will always choose 5 or lower, so any number that that max node finds will never be used. This technique is called alpha-beta pruning.



```
function ALPHA-BETA-SEARCH($state$) returns an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the $action$ in ACTIONS($state$) with value $v$

function MAX-VALUE($state, \alpha, \beta$) returns $a\ utility\ value$
  if TERMINAL-TEST($state$) then return UTILITY($state$)
  $v \leftarrow -\infty$
  for each $a$ in ACTIONS($state$) do
    $v \leftarrow$ MAX($v$, MIN-VALUE(RESULT($s$,$a$), $\alpha$, $\beta$))
    if $v \geq \beta$ then return $v$
    $\alpha \leftarrow$ MAX($\alpha$, $v$)
  return $v$

function MIN-VALUE($state, \alpha, \beta$) returns $a\ utility\ value$
  if TERMINAL-TEST($state$) then return UTILITY($state$)
  $v \leftarrow +\infty$
  for each $a$ in ACTIONS($state$) do
    $v \leftarrow$ MIN($v$, MAX-VALUE(RESULT($s$,$a$) , $\alpha$, $\beta$))
    if $v \leq \alpha$ then return $v$
    $\beta \leftarrow$ MIN($\beta$, $v$)
  return $v$
```

Alpha-beta algorithm (Stuart Russell & Peter Norvig, 2010, p.170)

When two AI played against each other with minimax or alpha-beta with the material evaluation, they kept playing the same moves over and over. They then got stuck in an infinite loop of each going back and forth between 2 different moves. This wasn't because there was anything wrong with the algorithm, but from the evaluation function, each side thinks that those repeated moves are the best moves they can make. Because of these ties happening so much, I made the program stop the game when a tie was identified. I defined ties in my program where no new pieces are captured by either side in 100 moves.

To try to solve this problem of ties happening, I made a set that contained previous states of the board that occurred in the game. If the AI had seen the board state two times or more, it wouldn't play a move, even if it thought it was the best one according to minimax/alpha-beta. This approached worked somewhat, although the AI just found new moves to go back and forth on. For example, instead of moving the rook back and forth, it would move the knight back and forth, and then move the queen back and forth once it couldn't move the knight back and forth anymore, etc.

Since the set of previous states didn't work, it seemed like the root of the problem was the evaluation function, and how it didn't account for positioning advantages. This meant the beginning of the game, where little to none of the pieces are taken, wasn't being quantified. If the AI could go up to maybe 10 or 20 depth, this would be less of a problem since it would see further into the game where more pieces could be taken. Since quantifying positional advantage well in chess is complex, I went for a simpler evaluation function which factored in how many legal moves the pieces on the board could make. For minimax, this was (number of legal moves white can make) – (number of legal moves black can make). I could then add this to the material score and return this as the evaluation function (material score + mobility score). All I needed then was a weight to put on the mobility score to make sure it didn't overpower the material score. This optimal weight for this (shown in results) for my AI was 0.35 (material score + mobility score * 0.35). When I refer to "mobility evaluation function" in this paper, I'm referring to the (material + mobility*weight) evaluation function.

Results & Discussion

Since minimax and alpha-beta should yield the same results, the first test that was done to make sure they were both working properly, was to play minimax vs alpha-beta at the same depth to verify they have a 50/50 win/loss ratio between the two. This was verified so it seems like alpha-beta and minimax are working fine for the AI.

Since minimax and alpha-beta are deterministic, if you run 2 AI vs each other on the starting position of chess, you will get the same results every time. For this reason, to get my results I had each player randomly move 5 times. Then I will check with my evaluation function to make sure the randomness didn't result in a super unfair board.

A total of 10 random moves were chosen because even though it checks with the evaluation function to make sure it's not too unfair, it may be unfair in ways the evaluation function doesn't see if there are more moves done. More moves done may result in less ties, which would help with results speed, but the results may be worse quality.

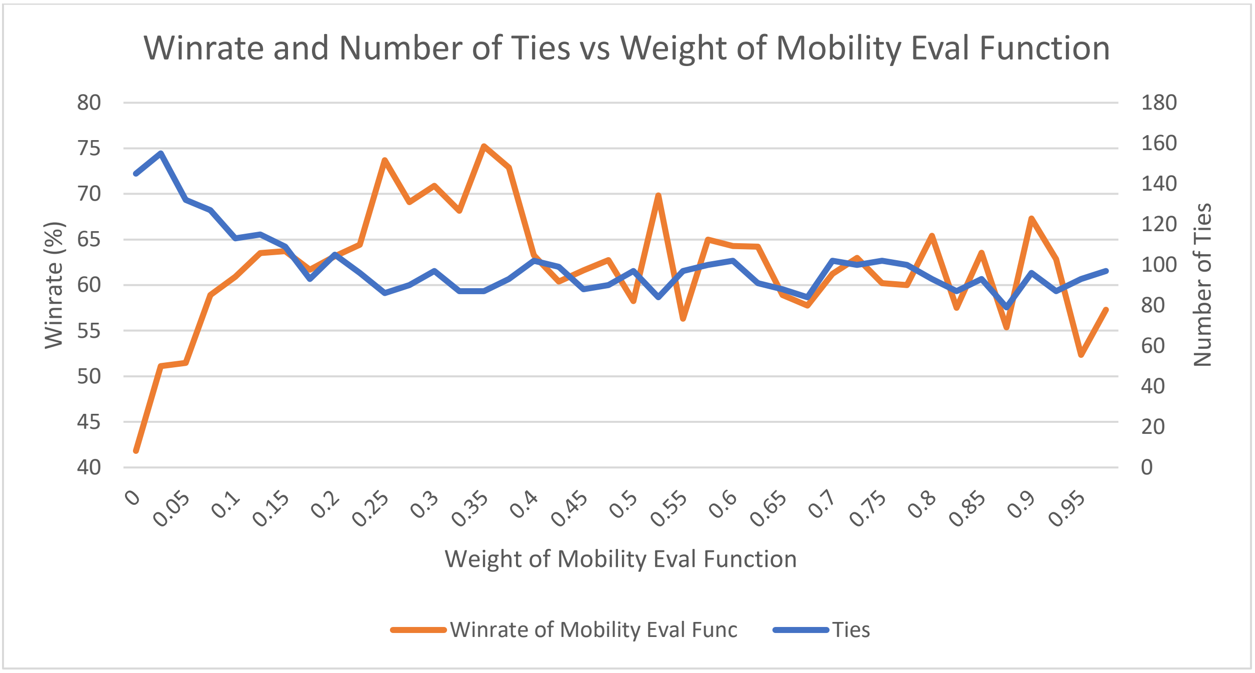
To determine the effects of depth on win rate, the AI was paired against different depths to play 50 games and the results were recorded. Since alpha-beta and minimax were proved to be equal with my AI earlier, alpha-beta was used for this table.

Win rate of AI at different depths using mobility evaluation function

	Depth 1	Depth 2	Depth 3	Depth 4	Depth 5
Depth 1	N/A				
Depth 2	98%	N/A			
Depth 3	100%	98%	N/A		
Depth 4	100%	100%	100%	N/A	
Depth 5	100%	100%	100%	97%	N/A

At just one depth above the other AI, the AI seemed to have near 100% win rate. Anything depth advantage above this didn't really make any difference since the win rate was already at 100%.

To determine what the best weight for the mobility evaluation function compared to the material evaluation function, I ran 200 games at each increment of 0.025 weight. I recorded the number of ties, as well as the win rate of the mobility vs the regular material evaluation function, both at depth of 3.

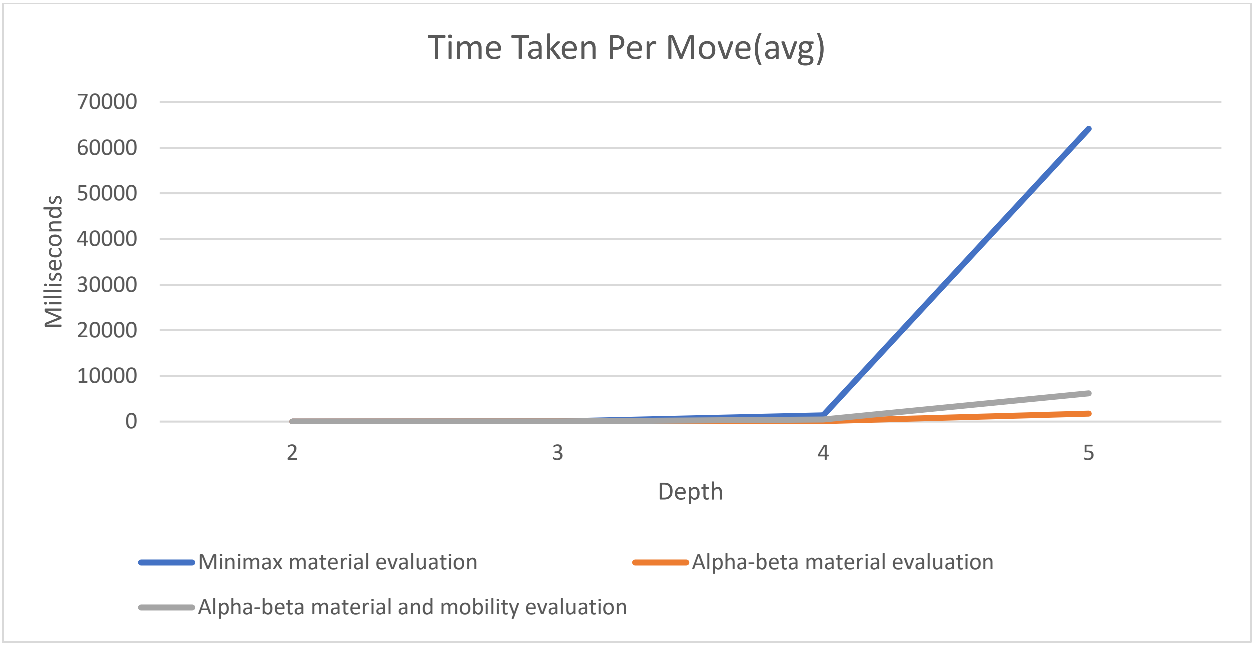


This new evaluation function was originally made to cut down the number of ties, but it also had the side effect of being a better evaluation function for winning. As the weight goes up, until about 0.35 weight, the win rate of the AI using the mobility evaluation function with the material evaluation function goes up. This works perfectly because around 0.35 weight is also when the number of ties stagnate at ~100.

Playing the material evaluation function vs the mobility evaluation function at a weight of 0.35, both at depth 4 (instead of depth 3) yielded a 95% win rate for the mobility evaluation function. This is interesting since with both at depth 3, there was only a 75% win rate. This probably means that the mobility evaluation function gets better with the more depth is provided to the minimax/alpha-beta algorithm.

To see how meaningful the win rate of 75% was compared to the win rates of different depths, I compared the evaluation function that included mobility at depth 3 vs just the material evaluation function at depth 4. Although the win rate of depth 3 vs depth 4 is usually near 0% for depth 3, the win rate that included mobility in the evaluation function was 25%. This is quite a big jump from the higher depths winning almost all of the games, to a 25% win rate even at a 1 depth disadvantage.

To make sure this mobility evaluation function isn't too slow compared to the material evaluation function, I ran timing tests at different depths comparing the two. We can also see the time difference between alpha-beta and minimax at similar depths.



In the graph there is a huge time penalty for not using alpha-beta over minimax. This is coupled with the fact that there is no downside to using alpha-beta since there is no information lost in the pruning steps. We can also see that the time taken between the material evaluation and the material and mobility evaluation isn't too large and is worth the win rate boost that comes along with using the mobility evaluation.

Although I am not particularly good at chess myself, I played vs the AI at a depth of 5 with alpha-beta and the mobility evaluation function and couldn't beat it. If I had more time, I would try to play some online chess games and see what rank my chess AI could get to. This would probably also help find any other big problems with the evaluation function.

Conclusion

The best the chess AI that was created could get is using alpha-beta with the mobility evaluation at depth 5. There are many more evaluation function upgrades that could be made, such as king safety, weighting pieces differently depending on the phase of the game, or pawn structure. For each of these evaluation function upgrades, the same process as finding the mobility weight could be done to find their best weights (although may get harder if they interact with each other).

Alpha-beta is a huge gain over minimax, but there are other search improvements that could be done. Implementing quiescence search would increase the win rate of the AI by expanding more nodes where pieces are getting captured. With enough upgrades of search or evaluation functions, an AI with a depth disadvantage should be able to win over 50% against an AI with regular material evaluation.

References

Russell, S. J., & Norvig, P. (2010). Artificial intelligence: a modern approach. Upper Saddle River: Prentice-Hall.