Atta Ebrahimi

Rebecca Hwa

CS 1571

10 October 2016

**Game of Amazons – Minimax Alpha-Beta Pruning**

The minimax algorithm is a recursive method for choosing the next move in an n-player game, usually a two player deterministic game. A value is associated with each position or state of the game. This value is computed by a heuristic function that it indicates how good it would be for a player to reach that position. The player then makes a move that maximizes the minimum value of the position resulting from the opponent’s possible moves. The algorithm can be thought of as exploring the nodes of a game tree. The effective branching factor of the tree is the average number of children of each node. The number of nodes increases exponentially as we increase the depth of the tree. From this, the number of nodes to be explored is approximately the branching factor raised to the power of the depth.

In the Game of Amazons starting with four queens per player and a 10x10 board, the branching factor starts out astonishingly high yielding 2176 moves (this number decreases to about 50 towards the end of the game as the board gets filled). In order to do calculations on a tree this large we require some sort of way to either minimize the moves or evaluate them in a time faster than bd. In this case, I used alpha-beta pruning in addition to minimax.

Alpha-beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. It will stop evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further because we would have no intention of picking them. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that could not possibly influence the final decision. With this we can avoid making calculations on large portions of the tree and reduce the runtime, although not asymptotically.

Although alpha-beta does save time, the major downfall of my project is the time limit. With 15 seconds per move it is hard to go as deep as I would like in order to make “well informed” decisions. My default depth is only 3 but due to the large branching factor in the beginning of the game I cannot make very good decisions early on. In order to prevent my turn being skipped I implemented a timer that will count to 14 seconds in the getMax and getMin functions, if the timer moves past 14 seconds it will return the best move found so far. This is not ideal, but it does produce a valid move and continues the game. There are a lot of things that go into why my move calculations are so long, one being the generation of successor states, and the second being my heuristic. To generate my successor states I convert the board into my own class which uses the numpy python package which makes array math easier. Generating the moves requires a large amount of computation due to python’s copy.deepcopy() which was my way of dealing with mutability of the board. My heuristic was also based on the amount of successor states that each possible move could generate, this in turn does the same amount of math and same amount of deep copies that my actual tree generation does which is very bad for the performance. Due to lack of time to come up with a work around for this, I opted to just leave it as is and use the timer that I made to prevent the move from going overtime although it could have been much faster.

I came up with two heuristics from the beginning that I thought would be pretty decent. Due to the fact that we are trying to maximize the area at the end of the game, I thought that the amount of space around a queen would be a decent heuristic. This makes the game choose the state that leaves as much space around the queens as possible. The other heuristic, and the one that I ended up using, was successor states from any given board. This seems to be directly correlated to the game winning condition which is the last person to move wins, therefore I opted to maximize my available moves on each turn. The benefit of this was that I already had a function that generated the amount of moves, all I had to do was take the length of that.

The results of my implementation on a 10x10 board with four queens and 15 seconds per turn are far less effective than I would expect. In the transcript provided of human vs ate9 I completely dominated it, winning by 63 points. This almost certainly has to do with the fact that it cannot produce a worthy move in 15 seconds. In the other two cases in the same file I produced a decent output, my game seems to back itself into a corner unnecessarily. The way my implementation plays against itself is also interesting as it tends to cover only one half of the board and the winner will end up winning by a large margin. It seems that I can beat it with only a little bit of effort, which is unfortunate. I think that if I implemented either a more powerful heuristic, or used a learning technique to store moves that resulted in a win I could improve on the current state of it quite a bit.