Spencer Carr & Ryan Skinner

Artificial Intelligence 4200.01

Dominick A. Atanasio

5 May 2019

The alpha beta pruning algorithm is an extension off of the minimax algorithm, and seeks to reduce the number of nodes that are explored, thus increasing the efficiency of the minimax algorithm. It accomplishes this by utilizing alpha and beta values that are passed through each recursive call. The alpha and beta values are used in a smart check in the program to see if the currents nodes children are worth evaluating. These checks eliminate useless branches, and the elimination of these branches is the essence of the algorithms optimality.

When implementing our alpha-beta pruning algorithm, our first step was to develop a heuristic that would be used by our evaluation function. After playing around with the game, the strategy that was most compelling seemed to revolve around limiting the opponents moves, and at the same time, keeping your move options flexible. It was also determined that between these two factors, keeping your move options flexible was of higher importance than limiting the opponents moves. These observations yielded the heuristic for the evaluation function to be the number of moves the computer has, multiplied by a certain weight, minus the number of moves the opponent has, multiplied by some weight, minus the number of surrounding moves, multiplied by some weight; the weights were determined after vigorous testing.

After we had this heuristic function, we implemented the evaluation function, that is, the alpha beta pruning algorithm. While implementing the algorithm, a few problems blossomed. These problems included memory space, and speed. Developing the algorithm in Java meant massive memory usage, and dealing with overhead. Some of the ways we dealt with these issues included, using a bitmap instead of a byte array, calling garbage collection manually, using smaller variable types, and storing moves. For the first idea, that is, the bitmap, we realized in the calculation for which moves are available, we were using an array of boolean values. We lowered our memory usage by using a bitmap instead of an array of boolean values. Another memory saving technique was simply calling garbage collection manually. Heap space was a valuable resource during the run of the program, especially when working in java, where everything is allocated on the heap. Another memory saving measure that was used was using smaller data types. So instead of ints and doubles, we used primitives such as bytes and shorts. In terms of efficiency, we stored previously made moves in a data structure, so when we came across the same position in different calculations, we could just reference memory, rather than running another calculation. Another measure that was taken to enhance efficiency was dynamically changing the depth throughout the program. We realized that during the initial stages that a high depth means long calculations, and that this depth isn't as important since its the initial stages of the game. We understood that the middle game and endgame is when the computational power is most needed. Realizing this, we had it so that as the program progressed further into the game, the higher its calculation depth would go.