Christian Heinrich

10/7/14

I opted to represent the state as four different arrays of length two each, one for each of the knights, with the first value in the array being the x of the corresponding Knight, and the second value being the y of the corresponding Knight. I chose to represent the states in this way as it was the easiest way to specify the starting positions of the Knights, as well as modify their position. For the successor function, I input the two knights of the current player, as well as the opponents knights, calculated the possible successors, and they if these were valid, appended them onto the end of a list and returned the list. I chose to do the successors this way in order to cycle through the various successors to determine the best of them, as this was the most efficient method to do so.

The Best_Move Method takes in seven arguments, which are in order: player 1's first Knight (pKnight1), player1's second Knight(pKnight2), the opponents first Knight (oKnight1), the opponents second Knight (oKnight2), the evaluation to use for Player 1 (pEval), the evaluation to use for the opponent(oEval) and the cutoff depth (cutoff). The first player will always be the first one to go, so to change which order the players go in, simply flip their pieces.

So for white to go first, enter best_move([3,3],[4,4],[3,4],[4,3],1,1,2)

And for black to go first enter best_move([3,4],[4,3],[3,3],[4,4],1,1,2)

The play function has two additional arguments, turn, which indicates whose turn it will be first, so if you want the player whose knights correspond to arguments 3 and 4 for play to go first, set turn to two (if set to anything else, it will automatically set turn to 1 in the first run through the loop). The final argument is printout, which, when set to 1, will output the moves made over the course of the game.

The two evaluation functions I opted to use were both based on distance, but a different form of distance for each. The first evaluation function was based on the distance between the players Knights and the top left corner (0,0). Using this evaluation function should prioritize making movements which will prioritize knights which are farther away from (0,0) than knights which are closer, with the goal of keeping both knights as far away from the origin as possible for as long as possible. The second evaluation function is based on the distance between a players two knights, and will prioritize the separation of the knights, in the hopes of minimizing knights getting clumped together and possibly trapped earlier than normal.

The result of these two evaluation functions is that the second evaluation function will generally be more successful than the first evaluation function, due to the greater emphasis on separation between knights. It will even regularly beat the brute force method, while being more efficient. Across a dozen games with knights in the positions [30,30], [40,40], [40,30] and [30, 40] the second evaluation function won 90% of the time over both the first evaluation function and the brute force method, with cutoff depths of ranging from 3-5. I initially thought this had something to do with an advantageous position, as I had set the second player to use the second evaluation method every time, but when I flipped it to having player one using the second evaluation function and player two using either the first evaluation or brute force, suddenly player one was winning over player two every time. Furthermore, the second evaluation method will generally lead to fewer states generated than either brute force or the first evaluation function.

Brute force search scales extremely poorly, for example, when testing from the same positon as above, i.e. [30,30],[40,40], [40,30] and [30,40] with both players using Brute force, the program simply hung for an extreme amount of time, to the point where I had to cancel the run.