Connect R

By: Casey Price

# Introduction

The purpose of this project was to develop a connect 4 AI using the minimax method by enumerating out all of the possible plays and picking the one that will result in the best board state for the player. For this project the AI had no longer than a set time to return its play so the depth of the tree the algorithm could build was limited by time constraints. The game also had to have a variable row and column count and variable win condition for how many you had to get in a row.

# Methods

I wrote my AI in C#, using the .NET framework and VS2014. I used WPF for my GUI, to create a Connect R style board that you can click on to play. When the user clicks a column then the board plays a piece in that column.

The way that I implemented my minimax tree was first creating a node object Called StateSpace that held a board, a list of sub boards that could be reached from that board in 1 play, and a fitness value for that specific path in the tree. I then created a recursive method that made every play possible from that that initial board state and set the list of children on the node. The function took in a parameter to signal how many plays ahead to look, which had to be tested in order to fulfil the timing requirements.

Once the tree was created I could begin applying my fitness function to it. You begin by doing a depth first search of the tree and then applying the fitness function to the leaf nodes of the tree. You then return the min or max of the nodes depending on whose turn it was, and alternate doing that all the way up the tree until you have the play that would ultimately yield you the best board state down the line. Obviously the algorithm can only look so far ahead so it is possible that it may miss an outcome or series of plays that it will lose to that take longer than the amount of plays it can look for, but that is part of the timing constraints.

The most complicated part of this project, by far, is the fitness function. Trying to analyze a Connect R board for who is winning in its current state is no easy task. I had lots of trouble with getting my AI to even realize when it could lose. I tried to simply teach my AI 2 things. First the center of the board is more valuable than the edges, and second that having more than 1 piece in a row that is not blocked from creating a winning row is a good thing. The more pieces you have in a row, the better your board state is. However if you have 2 or 3 pieces in a row that cannot be turned into a winning sequence because either it is blocked or does not have room to complete its sequence, then that sequence is essentially useless. I simply did a brute force check on my board, checking every single location for sequences that were going up, right, diagonally up, and diagonally down. I didn’t need to check left, down or the other directions because they would also inherently be checked by this method.

# Experiments

I did multiple experiments on my fitness function and it went through multiple iterations. The first iteration I simply added the fact that the center of the board is more valuable than the outer pieces. Obviously this just resulted in an AI that played in the center of the board and moved outward from there, pretty boring, and not very good at the game. Then I added the ability to spot check every single spot on the board for sequences. The more pieces in a sequence the more the sequence valued. The problem that stemmed from this algorithm is when you would have 2 pieces, then an empty spot, then another piece, the functions wouldn’t recognize this as a possible win, or as good as having 3 in a row. I had to check for empty spots in a row. Once I added that I came into the problem of the game thinking it could play off the board or not realizing a sequence was blocked. I had to change the algorithm again, this time making pieces that couldn’t be used to win because of space constraints worthless as well. There’s no value in having 3 pieces in a row if there’s an enemy piece on either side of it. Once I had those rules in place I thought of one that I figured out by playing the AI a lot and finding out ways to beat it. It’s that a piece that has more an empty spot on either side of it is more valuable than a piece that is blocked on one side and empty on another. For example if I have 3 in a row with an empty slot on both sides, then the opponent cannot block both and I am guaranteed a win. So the last rule I gave the function stated that pieces with empty spots next to them were more valuable than ones with none.

Once I had all of these rules in place, I was actually unable to beat my AI. Granted I am not the best connect 4 player, and didn’t try super hard, but the AI was still able to tie or beat me every time.

When playing around with the timing constraints, I initially had lots of problems. The first was again with cloning, which seems to be a theme with projects in this class. I quickly wrote a clone method to alleviate the problem. However, it was then taking over 3 seconds to enumerate an entire tree that was only 2 levels deep. This was completely unacceptable, and I eventually figured out the long enumeration time was due to the fact that I was cloning a user control that had Image objects inside of it for the GUI. Once I created a dummy object that only held a state to represent a square, cloning took considerably less time and the algorithm was taking less than a millisecond to enumerate a tree of 2 iterations.

I also added a little bit of Alpha-Beta pruning, if there is a column that is full I told the enumeration tree creator and also the parser to not bother to go down that path. This increases speed considerably when the board is almost full, but only when that is the case.

# Analysis

I ran multiple tests on my algorithm and have found the following data when it comes to timing.

As the size of the board increases, the time it takes to run the AI increases, due to the fact that there are more options and also more spots to check. Increasing the number of rows increases the runtime in a larger magnitude than increasing the columns.

The following data was found running the algorithm through 4 iterations and 7 Columns.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Rows | 5 | 10 | 15 | 20 |
| Time | 20.25 | 37.75 | 54.75 | 74 |

The following data was found running the algorithm through 4 iterations and 6 rows.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Columns | 5 | 10 | 15 | 20 |
| Time | 4.75 | 179 | 1253.75 | 5625.375 |

Increasing the number of iterations increases the time exponentially since every iteration creates exponentially more nodes to apply the fitness function to.

The following data was found running the algorithm through 6 rows and 7 columns.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Iterations | 1 | 2 | 3 | 4 | 5 |
| Time | 0.475 | 2.975 | 24.5 | 218 | 1725.875 |

# Conclusion

The minimax tree works very well for connect 4 because it is a zero sum game that has a very limited number of moves. Since any one move means the same plus or minus for both players, this makes it possible to develop a fitness function that can analyze the board from both states the same way. If the game were not to be a zero sum game, or say a move was only bad for one player, or worse for one player than it was good for another, then minimax would become impossible to use.

If I were to do this AI over again, I would probably do the fitness function the same way, but add in alpha-beta pruning and possibly be greedier with the time constraints. Currently I just hardcoded the iterations based off of the constraints given, and was safe with them to not go over, however I could possibly iterate more and look at more options if I had a running timer that automatically returned after the time limit rather than just safely going under the limit. I could’ve also added alpha-beta pruning to ignore plays/paths that were clearly either bad or stop when we found a win condition. Currently the only Alpha-Beta pruning I do is when there are columns that are full.

Another possibility would be to add threading to the tree iteration to apply the fitness function on multiple boards at once and also iterate multiple paths in the tree. Currently the slowest part of the AI is the fitness function and enumeration. If I could partially do those sections in parallel then this would increase the speed of my algorithm so that it could look further ahead and be a smarter algorithm.