Thomas the Stank Engine

A report on the famous* Connect 4 algorithm

Development Process

This engine was developed in Java using IntelliJ, using GitHub for version control and cloud storage. The debugging features in IntelliJ helped resolve some issues that would have been difficult to trace in a more barebones program. One such issue was every new node created in the game state tree using the same memory address, leading to every node having hundreds of moves being performed on it immediately. Having the ability to view the state of each variable and how it updated at different breakpoints made the problem pretty trivial to spot.

Search Algorithm

The standard minimax algorithm was used for this engine. It is based pretty closely on the algorithm described in the lecture notes. Other variants were considered, however once the evaluation function reached a satisfactory level, the engine was able to play the game well enough at low depths of minimax that it wasn't deemed necessary to look further into more efficient alternatives to minimax.

The main alteration that had to be made to the algorithm was the addition of checks for null nodes. In it's early stages, the engine had an issue where it would have a tendency to suggest making every move on either the leftmost or rightmost edge of the game board, even at depth 8. It would seem that part of the cause for this was that the search tree generated 7 children for each game state – one for each column – and didn't distinguish between legal and illegal moves. The evaluation function at the time, erroneously indicated that a move that didn't change the board state was actually really good, and so minimax determined that these outside edge moves, which got the engine closer to these 'ideal' moves, were the best way to win the game.

The null nodes were introduced to the search tree to prevent this tendency for illegal moves. When the search tree is generated, and the child nodes perform additional moves in the game, it checks if those moves are legal. If not, those children are set to null, and no further branches from that illegal game state are generated. Minimax, then, needs to check if a node is null, and therefore an invalid game state, before doing any processing on that node.

Evaluation Function Development

The evaluation function for this engine was developed from scratch, and some important lessons were learned along the way.

It's first iteration went through every cell on the board, and if a player had made a move there, it looked at two cells in each direction, to find 2-in-a-rows, 3-in-a-rows, and 4-in-a-rows. It was fairly time consuming as it had to look at 17 cells 42 times over, and additional checks had to be made to check for out-of-bounds cells, and to determine if it really found a 4-in-a-row and not just 4/5 cells in a row with one of the opponent's pieces in the middle. Also, since every 3-in-a-row was found from 3 different places, the evaluation scores were inflated well beyond what they were intended to be. It also attempted to track each player's scores separately, rather than to simply make player one positive and player two negative. Between issues in the function and the rest of the engine, it was unable to successfully identify ideal game states.

*: It's not actually famous

The second iteration of the evaluation function made fewer checks and had a more simplistic scoring system, but failed to address the fundamental issues of the first iteration. It still attempted to track which player it was evaluating for, and at one point caused every move to be optimised for player 1, leading to a strange cooperative approach when playing against itself, trying to get player 1 to have as many high-scoring 4-in-a-rows as possible. However, this seemed to be in part due to coincidence, as small adjustments to the engine stopped this cooperative behaviour.

The third and final iteration finally evaluated game states successfully. It removed a lot of unnecessary checking of cells, and most importantly correctly assigned positive scores for game states favouring player 1 and negative scores for those favouring player 2. It also more rigorously ensured that points would not be assigned for 2-in-a-rows and 3-in-a-rows if they could not be turned into a 4-in-a-row.

The scoring scheme for this final iteration is as follows;

- Sets of 4 cells in horizontal, vertical, and diagonal rows are tested
- **10 points** for each potential 4-in-a-row where a player has **one** piece and the other 3 positions are open
- **100 points** for each potential 4-in-a-row where a player has **two** pieces and the other 2 positions are open
- **1000 points** for each potential 4-in-a-row where a player has **three** pieces and the other position is open
- **10000 points** for each completed 4-in-a-row
- These points are **positive** if owned by player 1, and **negative** if owned by player 2

Win, loss, and draw results are not explicitly tested for beyond these scores – the heavy weighting towards 4-in-a-rows means that the engine will successfully favour moves that lead towards 4-in-a-rows as well as those that avoid a 4-in-a-row being created by an opponent. If no more potential 4-in-a-rows exist, the evaluation function should provide a score of zero.

In earlier iterations the evaluation function assigned more points for open positions that were currently playable, and even a variant of the final function attempted this. However, not only did it make the evaluation function more convoluted, it also made it more difficult for the scoring system to correctly favour good moves.

Final Performance

The engine, in it’s current state, is able to decide on effective moves with a minimax depth of only 4. Due to three-in-a-rows having a high evaluation score, it is able to set up threats even if it is unlikely to be able to capitalise on them in the next 4 moves, and is able to correctly respond to threats as well. It also calculates these moves quite quickly – on my i5-6600K running at 4GHz it responds to the **go** command without a noticeable delay.

Earlier in testing I hardcoded the first move to play in the centre column with an arbitrary score, and assigned extra points for making plays in the middle, only for the engine to continue favouring the edges. Now, it is able to come to those conclusions without interference. I have left the first move hardcoded, as it is marginally more efficient. The hardcoded score is the expected result from the evaluation function.