Ultimate 4x4 Tic Tac Toe Bot

Arjun P, Sai Anurudh Reddy Peduri

1 Search Algorithm

We used min-max with alpha-beta pruning and iterative deepening starting with a max depth of 3.

We run the search for thirteen seconds and then abort and play the best move according to the last fully completed search.

2 Heuristic

The board is the entire 16x16 game board. A block is a 4x4 subgrid. A cell is a single (1x1) cell.

2.1 Block Score

For each block, we compute the block scores from our perspective and from the opponent's perspective separately.

The score of a block is $score_{block}$ if we won and 0 if the opponent won. Otherwise, the score of the block is the sum of the scores of its cells.

2.2 Cell Scores

Consider a set of four cells forming a row, column or diagonal. Let's call this set a 1-attack if we can win it in one move, 2-attack if we can win it in two moves, and 3-attack if we can win it in three moves.

If a cell is part of a 1-attack, we give it score $score_1$.

If the cell is not part of a 1-attack, let $count_2$ be the number of 2-attacks it is part of for us and $count_3$ be the number of 3-attacks it is part of. Similarly opp_2 and opp_3 for the enemy. The score of the cell is:

$$my\ cell\ score = score_2 \cdot (my\ count_2)^2 + score_3 \cdot (my\ count_3)^2$$

2.3 Attack Score

$$attack\ score = \sum_{blocks} (my\ block\ score - opp\ block\ score)$$

It represents the overall "goodness" in all blocks. It roughly is an estimate of how good we would do if the game ends up drawing.

2.4 Game Score

We calculate game score separately from our perspective and from the opponents perspective.

The game score represents the chances of outright winning the game. Consider the `block_status`, which is a 4x4 grid representing which blocks have been won, lost, drawed, or not decided yet.

Here again we have 1-attacks, which are sets of four blocks such that we can win the game by winning one more block. And similarly, 2-attacks and 3-attacks.

Find the score of each block in `block_status` just like how we found cell scores before. Take the weighted sum of these scores, where the weights are the the *my block score* values computed before. Basically we are giving more weight to blocks which are more likely to be won.

The shortfall of this is if we have four blocks in a row which are all highly likely to be won but have not yet been won, the heuristic won't notice at all that this is better than having those same four high chance blocks randomly scattered around the board. This is because until we win the block it doesn't contribute to the `block_status`. We did not have time to overcome this issue.

2.5 Heuristic Score

The final score given to a game state is:

$$heuristic\,score = weight_{attack} \cdot (attack\,score) + (my\,game\,score - opp\,game\,score)$$

2.6 Parameters

In this discussion, $score_{block}$, $score_1$, $score_2$, $score_3$, and $weight_{attack}$ are parameters. We went with

$$score_{block} \gg score_1 \gg score_2 \gg score_3$$

and

$$weight_{attack} = \frac{score_1}{score_2}$$

.

We considered optimizing these parameters using a random restart hill climb search but that would require too much computation time.

3 Optimizations

Instead of recalculating all the block scores for every turn, we need only recalculate for the block that was played in in that turn.

Also, we manually did some performance optimizations such as loop unrolling and inlining some functions, which proved to make a huge difference. In a language more suited for such tasks, these optimizations would have been performed by the compiler.