3D Tic Tac Toe

Adina Rubinoff
CSC 242
Febuary 7 2010

The goal of this project is to provide a Java program that plays a game of 3D 4x4 Tic Tac Toe . The program uses a minimax algorithm with optional alpha-beta pruning. In addition to specifying whether alpha-beta pruning is used or not, the user may also choose among a few different evaluation functions, and specify the number of turns ahead the program looks. Also provided is a python script for pitting two different computer-controlled players against each other.

Algorithms

This program makes use of the minimax algorithm with optional alpha-beta pruning. The board is represented as a one-dimensional array. A variety of different evaluation functions are provided. Also included is a function for checking if the game has been won.

Board State Representation

To represent the board, this program uses a 1x64 array of integers. The player's pieces are represented with 1s, the opponent's pieces with -1s, and blank spaces with 0s. With this representation, position x,y,z in the 3D board is located at position x+4*y+16*z.

Win Detector

To detect a win, this program simply looks at every possible row, column and diagonal to see if any one player's pieces occupy all 4 spaces. This is achieved using a series of conditionals. There are 76 different possibilities for a win (16 rows in each direction, 2 diagonals per face in each direction (which makes 12 faces), and then 4 corner-to-corner diagonals), and this algorithm checks each one in sequence.

Static Evaluators

This program provides four different static evaluators. Each one begins by looking at every row, column, and diagonal in turn. If both players have pieces in that row, it is ignored; if only one player has pieces in it, the evaluator counts these n pieces. At the end of this tally, the evaluator has a list of how many 1-in-a-rows, 2-in-a-rows, 3-in-a-rows and 4-in-a-rows each player has. Where the evaluators differ is in what they do with this information, as described below.

Eval1

This was the initial static evaluator first used in testing. It assigns a positive value to every n-in-a-row the player has, and a negative value to every n-in-a-row the opponent has. An n-in-a-row is worth about an order of magnitude more than an (n-1)-in-a-row. In addition, the opponent's rows are worth slightly more than the player's rows. This is to encourage the player to block before building its own rows.

Eval2
This evaluator is almost identical to Eval1; it only differs in the values assigned to the various n-in-a-rows. This evaluator makes sure that the value assigned to an n-in-a-row is greater than 76 times the value assigned to an (n-1)-in-a-row. This is to ensure that no matter how many (n-1)-in-a-rows a player (or its opponent) gets, it still considers an n-in-a-row more important.

EvalSame
This evaluator is the same as Eval1 except it assigns the same values to the player's and the opponent's n-in-a-rows. This is for testing purposes, to see if making the opponent's n-in-a-rows more valuable actually makes a difference.

EvalAbsolute
This evaluator is written on a different philosophy than the other three. Rather than assigning points for each n-in-a-row, it just returns a flat value based on who has more of the largest n-in-a-rows (positive if it's the player, negative if it's the opponent).

Minimax Algorithm
This is the algorithm that allows the player to look ahead to future moves and pick the move that gives him the best options for future moves. It has two main functions, the minmove and maxmove functions, which are mutually recursive. Given a set of potential moves, the player considers what the opponent's best move would be in each of those potential states (using the minmove function), and then chooses the move that gives his opponent the worst options. Similarly, the minmove function considers all the potential moves from a given state, considers what the player's best move would be from those (using maxmove), and returns the value of his worst option. This mutual recursion continues until the depth limit is reached, at which point the function simply evaluates all the potential states (using one of the static evaluators).

Alpha-Beta Pruning Algorithm
This algorithm is basically a fancier version of minimax. It calls minmove and maxmove exactly as described above, but also passes around two variables, alpha and beta. Alpha, initially set to -650001 (one less than the value of the worst possible move, one where the opponent wins), keeps track of the best move the opponent will let us get so far. If the minmove function encounters a worse move than alpha, it can quit right away, since it knows that the player can do better. Similarly, beta, initially set to 600001 (one more than the value of a win for the player), keeps track of the worst move the opponent can make the player take. If the maxmove function encounters something better, it can quit right away, since it knows the opponent will never let it take that move. Together, these variables allow the algorithm to examine many fewer nodes, thus speeding up performance considerably.

Results and Statistics

Comparing Static Evaluators

In order to determine which evaluator had the best performance, they were pitted against each other in the combinations documented below. Each contest consisted of twenty games, run with a lookahead level of three.

Eval1 vs. Eval2
Verdict: about equally matched
Eval1: 11 wins
Eval2: 9 wins

Eval2 vs. EvalSame
Verdict: about equally matched
Eval2: 11 wins
EvalSame: 9 wins

Eval2 vs. EvalAbsolute
Verdict: Eval2 is much better
Eval2: 20 wins
EvalAbsolute: 0 wins

Comparing Lookahead Levels

In order to determine the effect of lookahead levels on performance, players with different levels of lookahead were pitted against each other. The evaluator Eval2 was used for all players. As above, each contest consisted of 20 games. The results are as follows:

1ply vs. 2ply
Verdict: about equally matched
1ply: 11 wins
2ply: 9 wins

2ply vs. 3ply
Verdict: about equally matched
2ply: 10 wins
3ply: 9 wins
Ties: 1

3ply vs. 4ply
Verdict: about equally matched
3ply: 11 wins
4ply: 9 wins

Conclusions
It seems like altering the lookahead does not significantly affect the player's ability to win the game. Perhaps it would start to make more of a difference as the lookahead

increased. However, since we are limited to 5 seconds per turn, 4 ply is about the limit of how far my player can look.

Comparing Minimax and Alpha-Beta

Minimax: 9 wins
Alpha-Beta: 10
Ties: 1

This shows that the alpha-beta pruning doesn't affect the performance of the program, which is as expected, since it is supposed to be a speed improvement. To verify this claim, we tested the following statistics:

Branching factor:

	Minimax	**Alpha-Beta**
Average branching factor per move over an entire game	22.41176471	17.15384615
	20.63636364	26.1
	20.75	15
	17.10526316	20.77777778
	21.08333333	23.4
	21.07142857	17.22222222
	14.86956522	21.28571429
	18.53846154	14.8
	18	20.1875
	21.85714286	13.12121212

So alpha-beta pruning does have some positive impact on the game's average branching factor, although it may be slight. Per move, improvement can also be seen, as shown in the following chart:

MINIMAX			**ALPHA-BETA**		
Move Time	Evals	Avg. B.F.	Move Time	Evals	Avg. B.F.
0.173	13934	22.86875	0.155	5386	29.52879581
0.09	44944	34.53502235	0.047	21639	29.07244502
0.031	15424	18.83640553	0.032	16492	30.08787346
0.018	8863	14.84937888	0.042	21638	24.50487541
0.064	32111	30.13496377	0.016	8607	23.21025641
0.013	5690	28.73913043	0.021	10727	23.88747346
0.009	3637	7.434554974	0.019	8867	18.65940594
0.007	3074	16.00961538	0.015	6979	16.08154506
0.02	9620	21.01242236	0.014	6634	22.20952381
0.019	9002	24.19487179	0.035	14678	11.92210682
0.028	12311	17.99174691	0.049	21428	15.48785425
0.014	6604	21.63354037	0.038	15692	13.79105691
0.027	11969	15.48371532	0.045	19858	19.57796452
0.025	11836	15.65679012	0.024	9713	12.49764151

0.011	5649	13.35434783	0.022	9192	12.63556116
0.016	7990	14.15081967	0.012	5650	14.06436782
0.014	7325	13.68103448	0.009	4127	13.33827893
0.006	2690	10.40484429	0.009	4037	14.82993197
0.004	2050	12.86285714	0.003	1281	12.17948718
0.006	2568	11.32669323	0.006	2755	9.527607362
0.003	1705	15.1557377	0.005	1574	10.12571429
			0	125	10.12571429
0.02847619	**10428.38095**	**18.11034488**	**0.028090909**	**9867.227273**	**17.60661279**
Avg. Move Time	Avg. Evals	Avg. B.Fs.	Avg. Move Time	Avg. Evals	Avg. B.Fs.
MINIMAX			ALPHA-BETA		

This shows that the alpha-beta pruning does improve the amount of the graph explored. The number of evaluations, the average move time, and the average branching factor all drop. The drop is slight, but surely more improvement would be seen with a higher lookahead (these were all tested on a lookahead of 2).

Future Work

In the future, this project could be improved upon mainly by writing better evaluator functions. The current ones still allow the opponent to win sometimes, by failing to block, which could obviously use improvement. The project could also explore improvements to the alpha-beta pruning algorithm, to make the entire thing faster, which will in turn allow it to run at higher lookahead levels.