3D Tic Tac Toe
Adina Rubinoff
CSC 242
February 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:

<table>
<thead>
<tr>
<th></th>
<th>Minimax</th>
<th>Alpha-Beta</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average branching factor per move over an entire game</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>22.41176471</td>
<td>17.15384615</td>
</tr>
<tr>
<td></td>
<td>20.63636364</td>
<td>26.1</td>
</tr>
<tr>
<td></td>
<td>20.75</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>17.10526316</td>
<td>20.77777778</td>
</tr>
<tr>
<td></td>
<td>21.08333333</td>
<td>23.4</td>
</tr>
<tr>
<td></td>
<td>21.07142857</td>
<td>17.22222222</td>
</tr>
<tr>
<td></td>
<td>14.86956522</td>
<td>21.28571429</td>
</tr>
<tr>
<td></td>
<td>18.53846154</td>
<td>14.8</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>20.1875</td>
</tr>
<tr>
<td></td>
<td>21.85714286</td>
<td>13.12121212</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Move Time</th>
<th>Evals</th>
<th>Avg. B.F.</th>
<th>Move Time</th>
<th>Evals</th>
<th>Avg. B.F.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.173</td>
<td>13934</td>
<td>22.86875</td>
<td>0.155</td>
<td>5386</td>
<td>29.52879581</td>
</tr>
<tr>
<td>0.09</td>
<td>44944</td>
<td>34.5350235</td>
<td>0.047</td>
<td>21639</td>
<td>29.07244502</td>
</tr>
<tr>
<td>0.031</td>
<td>15424</td>
<td>18.83640553</td>
<td>0.032</td>
<td>16492</td>
<td>30.08787346</td>
</tr>
<tr>
<td>0.018</td>
<td>8863</td>
<td>14.84937888</td>
<td>0.042</td>
<td>21638</td>
<td>24.50487541</td>
</tr>
<tr>
<td>0.064</td>
<td>32111</td>
<td>30.13496377</td>
<td>0.016</td>
<td>8607</td>
<td>23.21025641</td>
</tr>
<tr>
<td>0.013</td>
<td>5690</td>
<td>28.73913043</td>
<td>0.021</td>
<td>10727</td>
<td>23.88747346</td>
</tr>
<tr>
<td>0.009</td>
<td>3637</td>
<td>7.434554974</td>
<td>0.019</td>
<td>8867</td>
<td>18.65940594</td>
</tr>
<tr>
<td>0.007</td>
<td>3074</td>
<td>16.00961538</td>
<td>0.015</td>
<td>6979</td>
<td>16.08154506</td>
</tr>
<tr>
<td>0.02</td>
<td>9620</td>
<td>21.01242236</td>
<td>0.014</td>
<td>6634</td>
<td>22.20952381</td>
</tr>
<tr>
<td>0.019</td>
<td>9002</td>
<td>24.19487179</td>
<td>0.035</td>
<td>14678</td>
<td>11.92210682</td>
</tr>
<tr>
<td>0.028</td>
<td>12311</td>
<td>17.99174691</td>
<td>0.049</td>
<td>21428</td>
<td>15.48785425</td>
</tr>
<tr>
<td>0.014</td>
<td>6604</td>
<td>21.63354037</td>
<td>0.038</td>
<td>15692</td>
<td>13.79105691</td>
</tr>
<tr>
<td>0.027</td>
<td>11969</td>
<td>15.48371532</td>
<td>0.045</td>
<td>19858</td>
<td>19.57796452</td>
</tr>
<tr>
<td>0.025</td>
<td>11836</td>
<td>15.65679012</td>
<td>0.024</td>
<td>9713</td>
<td>12.49764151</td>
</tr>
</tbody>
</table>
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.