# Autonomous Agents PLH412 Project

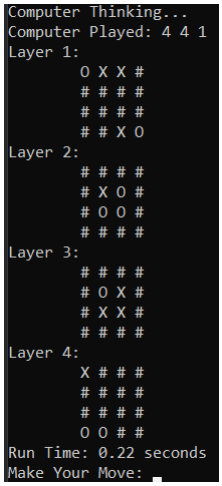
Diamantis Rafail Papadam - 2017030044

Monday 4th April, 2022

# Contents

# The 3D Tic-Tac-Toe

## Appearance of the Game




## Tic Tac Toe Notation

There are 3 possible symbols in any given cell:

- X: The pieces of the first player.
- O: The pieces of the second player.
- #: An empty cell.

## How to Play

The way to play, since there isn't a convenient GUI, is by typing 3 numbers every time that your turn comes. All of the numbers should belong in the set $\{1, 2, 3, 4\}$ and correspond to an empty cell, otherwise you will be asked to try again. The first number refers to the layer in which you want to play, while the second and third numbers correspond to the row and column respectively.

# Functions Used as Building Blocks

## Introduction

The functions that are described in this section, are very frequently used inside larger, more complex routines, that serve a wider purpose; in fact, both of our agents need their utility. Therefore, it's very reasonable that before we move into the specifics of the algorithms that our agents use, we should first understand what tools we have in our disposal. It's worth mentioning that because of the small size of the board, all the functions presented here, have a constant time complexity $O(1)$ and will be used as components to help us "build" more complex algorithms.

## Moves Based on Theory

The *get_theoritical_move(board)* function returns a random theoretical move as long as we're at the beginning of the game, more specifically, when 4 or less moves have been player so far. Theoretical moves are defined in the *structures.cpp* file as a vector of 8 moves that are considered the best options in the beginning of any 4x4x4 tic tac toe game. It is very useful for any agent to start with some theory because in the very beginning of the game, exploring different positions in low depth relative to the state space of the game won't really be of much use to him.

## Retrieve Available Moves

The function that manages to return a list of all the possible moves is called *available_moves(board)* and it simply iterates through all 64 cells to check which ones are empty.

## Player Win Check

Two functions were implemented with the purpose of checking if a specific player won. The first function is *playerWin(board, player)* and it actually uses 2 sub-routines to determine whether the specified player has won the game. Those sub-routines are *playerWinLayer(board, player)* & *playerWinInterLayer(board, player)*; the former iteratively checks 40 possible winning lines corresponding to all the winning combinations in each layer separately, while the later iteratively checks the rest 36 possible winning lines that can exist between different layers. The second function that was used instead, namely *playerWinFast(board, player)*, is actually "hard-coded" so that slightly less time is needed for it to run and hence, the performance of our agents improves.

## Game Over Check

The function that checks whether the game is over is called *game_over(board)*. It begins by calling *PlayerWinFast(board, player)* twice, to see whether either player won the game. If this isn't the case, it checks if there are no valid moves left in the game (by calling the previously presented function *available_moves(board)*), which would mean that we have a draw situation.

# Minimax

## The Standard Form of Minimax

In this section, we are going to show the structure of our algorithm in pseudo-code, as well as the 2 different evaluation functions that were implemented.

### Structure

```
int minimax(board, depth, player, eval_function) {
    if(game_over(board)) {
        if(draw)
            return 0;
        return (x won) ? infinity : -infinity;
    }
    else if(max depth is reached)
        return evaluation(board);

    if(x plays) {
        score = -infinity;
        for(move in available_moves(board)) {
            new_board = play_move(board, move);
            current_score = minimax(new_board, depth + 1, false, eval_func);
            score = max(score, current_score);
        }
    }
    else { //o plays
        score = infinity;
        for(move in available_moves(board)) {
            new_board = play_move(board, move)
            current_score = minimax(new_board, depth + 1, true, eval_func);
            score = min(score, current_score);
        }
    }
    return score;
}
```

The minimax structure presented above is slightly simplified and an especially important thing to point out, is that in our program, not only an integer value representing the score is returned, but also the move that corresponds to that score.

The initial call of minimax is always in the form *minimax(board, 0, player, eval_function)*, where *board* is the current position, *player* is a boolean value where true and false correspond to the X and O players respectively. Finally, *eval_function* is a pointer to the function which will be used by the algorithm to determine how good any given position is, when the maximum depth is reached.

## First Evaluation Function

For each one of the 76 possible winning lines, the first evaluation function begins by counting how many *X's* and *O's* can be found. Then in calculates 2 scores, *X_score* and *O_score*, with the following rule:

- 0 occurrences in a winning line $\rightarrow$ score = 0.
- 1 occurrence in a winning line $\rightarrow$ score = 0.
- 2 occurrences in a winning line $\rightarrow$ score = 2.
- 3 occurrences in a winning line $\rightarrow$ score = 6.
- 4 occurrences in a winning line $\rightarrow$ score = infinity.

All these scores are added up, separately for X and O. Finally, the evaluation function returns $(X\_score - O\_score)$, since X is the maximizing player.

## Second Evaluation Function

This evaluation function, is a slight modification of the first one. The core idea is the same with the difference that if on a single line there are 2 or 3 pieces of one player and even a single piece of the other player, then the score for both players will actually be 0 since there cannot be a tic tac toe there. To be more strict, for each one of the 76 lines:

- 2 X's & 0 O's $\rightarrow$ X_score = X_score + 2.
- 3 X's & 0 O's $\rightarrow$ X_score = X_score + 6.
- 4 X's $\rightarrow$ X_score = infinity.
- 2 O's & 0 X's $\rightarrow$ O_score = O_score + 2.
- 3 O's & 0 X's $\rightarrow$ O_score = O_score + 6.
- 4 O's $\rightarrow$ O_score = infinity.
- Anything Else $\rightarrow$ Both scores remain the same.

Intuitively speaking, the second evaluation function seems to be a more accurate option. Whether this is the case or not, remains to be seen in section Evaluation Function Comparison.

## a-b Pruning

The a-b pruning method, is probably the most widely known optimization of the minimax algorithm. In the best case scenario it can help explore twice as deep, in the same amount of time, while in effect it practically always allows us to search significantly deeper by pruning off sub-trees. The idea here is that more information and hence more intelligence is added into the algorithm. The simple explanation for this a-b pruning strategy, is that any node at any given time has the information of the highest and the lowest evaluation, that is possible from the previously calculated positions. This means that if we ever get $b \leq a$ at any node, the other player would have been able to force a better position for him earlier on. Knowing this, we can just prune the rest of the children of the current node and simply return to our parent with nothing changing. We are not going to get into more detail here, but should you be particularly interested in this subject, here is a more comprehensive source [1].

## Move Sorting

Sorting the moves from seemingly best to seemingly worst with a heuristic function is actually an idea that really boosts the performance of a-b pruning, as well as the performance of Forward Pruning. In fact, if the moves are somehow perfectly sorted from best to worst, we will have the best case senario for a-b pruning minimax, as mentioned earlier, and it will be able to search twice as deep compared to the standard minimax for a fixed amount of time; or equivalently, search exponentially faster for the same for a fixed depth. The heuristic function that was used for the move sorting is actually the second evaluation function. A C++ comparator was then used, depending on whether the X player is playing or not, and the moves were sorted in the according order. If X is playing, moves that return a higher evaluation are tested first, whereas if O is playing, moves that return a lower evaluation are examined first.

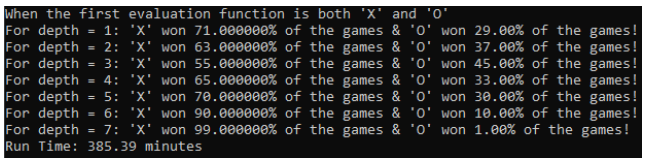
## Forward Pruning

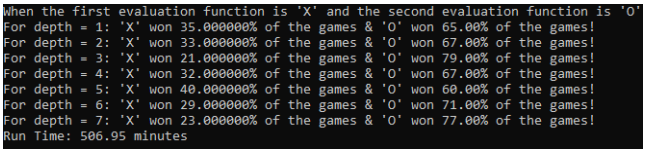
Forward pruning is the idea of examining less moves as the depth gets high and searching in it becomes time consuming and computationally expensive. In our program, for depths 0, 1, 2 there was no pruning at all. Pruning started on depth 3 and went up to depth 6, when in fact for depths 5 and 6 we basically had singular extensions, meaning that only 1 move was examined. The function that was used for pruning is: $f(x) = max(1, 64 - e^d)$, where d is the depth, x

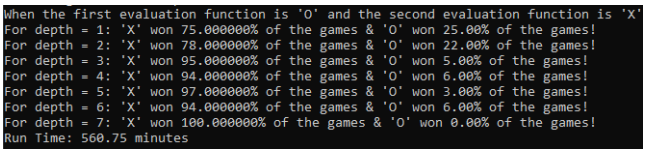
is the amount of moves and the result *f(x)* is the new amount of moves. After this calculation, the first *f(x)* moves, on the sorted move vector, were explored.

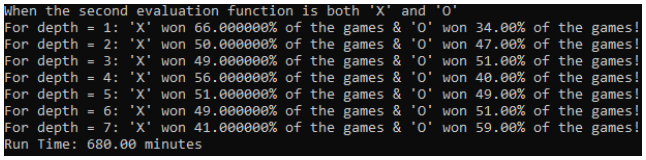
# Evaluation Function Comparison

## Images




## Tables

For the tables below we are going to use the notation:

- First Evaluation Function →EF1
- Second Evaluation Function →EF2

| TABLE 1 | | | |
|---|---|---|---|
| X Player - EF1 \| O Player - EF1 | | | |
| Depth | X win ratio | O win ratio | Draw rate |
| 1 | 71% | 29% | 0% |
| 2 | 63% | 37% | 0% |
| 3 | 55% | 45% | 0% |
| 4 | 65% | 33% | 2% |
| 5 | 70% | 30% | 0% |
| 6 | 90% | 10% | 0% |
| 7 | 99% | 1% | 0% |

| TABLE 2 | | | |
|---|---|---|---|
| X Player - EF1 \| O Player - EF2 | | | |
| Depth | X win ratio | O win ratio | Draw rate |
| 1 | 35% | 65% | 0% |
| 2 | 33% | 67% | 0% |
| 3 | 21% | 79% | 0% |
| 4 | 32% | 67% | 1% |
| 5 | 40% | 60% | 0% |
| 6 | 29% | 71% | 0% |
| 7 | 23% | 77% | 0% |

| TABLE 3 | | | |
|---|---|---|---|
| X Player - EF2 \| O Player - EF1 | | | |
| Depth | X win ratio | O win ratio | Draw rate |
| 1 | 75% | 25% | 0% |
| 2 | 78% | 22% | 0% |
| 3 | 95% | 5% | 0% |
| 4 | 94% | 6% | 0% |
| 5 | 97% | 3% | 0% |
| 6 | 94% | 6% | 0% |
| 7 | 100% | 0% | 0% |

| TABLE 4 | | | |
|---|---|---|---|
| X Player - EF2 \| O Player - EF2 | | | |
| Depth | X win ratio | O win ratio | Draw rate |
| 1 | 66% | 34% | 0% |
| 2 | 50% | 47% | 3% |
| 3 | 49% | 51% | 0% |
| 4 | 56% | 40% | 4% |
| 5 | 51% | 49% | 0% |
| 6 | 49% | 51% | 0% |
| 7 | 41% | 59% | 0% |

## Result Interpretation

First of all, it should be noted that this evaluation function testing was done with the optimized version of minimax, meaning that all the enhancements presented in this chapter have been added to the algorithm before the 2 evaluation heuristics were plugged in as a parameter and compared.

Notice that in tables 1 and 4, both players use the same evaluation function. In the case of EF1, we see that the player with the "X" pieces has a significant advantage, whereas in the case of EF2, the result is close "50-50". The reason we did those 2 tests was to understand whether the first player in a 4x4x4 tic tac toe game has an advantage over the second player. Unfortunately, we didn't

get consistent results to be able to come to a conclusion, but it has in fact been shown that the first player can force a win [2].

In tables 2 and 3 we see the actual comparison between EF1 and EF2. It is pretty clear from the two tables that the second evaluation function is much better than the first one and in fact, it beats it consistently for all depths. Additionally, as the depth of minimax increases, the superiority of EF2 over EF1 seems to be even more clear.

# Monte Carlo Tree Search

## Selection

When selecting a node down the tree there are 2 things that we need to consider. Firstly, how promising the node looks, i.e. how high the win ratio is so far. Secondly, we need to look at how few simulations we have done thus far, since a low number of simulations might mean that the node could in fact be promising and we haven't still figured it out. This dilemma is known as the exploration vs exploitation, where exploration refers to picking nodes with low number of simulations and exploitation refers to picking nodes with high win ratio. Thankfully, a group of researchers have already discovered a very efficient function [3] for the purpose of balancing exploration and exploitation.

## Expansion

After a node has been selected for expansion, there are two possibilities. If the node has 0 simulations so far, then it won't get further expanded quite yet and it will be the node on which the simulation will take place. On the other hand, if the selected node has already produced a simulation, it will be further expanded and a random child will be selected for simulation. In our 4x4x4 tic tac toe the expansion is as simple as *retrieving the available moves*, doing the proper initializations for every new node (representing a move) and then randomly selecting which one is to be simulated.

## Simulation

Having selected and expanded the proper node, now comes time for the simulation. This step is as simple as randomly making moves for each player until the game is finished with either player winning or a draw. This may sound strange for someone that's new to the Monte Carlo Tree Search (MCTS) algorithm, but in fact we rely on randomization in order to slowly converge to an optimal move, or in a more practical situation where we don't have infinite time in our disposal, to find a quite solid move.

## Backpropagation

This is the final step of the MCTS algorithm. After the simulation has been finished, we can take the result and backpropagate it all the way to the root of the tree. If we represent the root as $R$ and the simulated node as $C$, then we can denote the path as $C \rightsquigarrow R$. A noteworthy fact is that if on level $d \in \{1, 2, ...\}$ we have nodes representing the X-Player, then on levels *d-1* and *d+1* we must have nodes representing the O-Player and vice versa. Before we get in the backpropagation specifics let's denote the nodes that represent the X-Player on our path as $X_v$ and the nodes that represent the O-Player on our path as $O_v$. Finally, it should be mentioned that every node keeps track of the number of wins and the numbers of simulations with the variables $w$ and $n$ respectively. There are 3 possible outcomes, given the result of the simulation, for all nodes in the path $C \rightsquigarrow R$:

- X Wins $\rightarrow$ $X_v.w = X_v.w + 1 \mid X_v.n = X_v.n + 1 \mid O_v.n = O_v.n + 1$
- O Wins $\rightarrow$ $O_v.w = O_v.w + 1 \mid O_v.n = O_v.n + 1 \mid X_v.n = X_v.n + 1$
- Draw $\rightarrow$ $X_v.w = X_v.w + 0.5 \mid X_v.n = X_v.n + 1 \mid O_v.w = O_v.w + 0.5 \mid O_v.n = O_v.n + 1$

## Best Move

Below, we are going to present 3 different ways to evaluate the best move that can be played from the position that is described by the root of the tree $R$. It ought to be mentioned, that the evaluation we went with in our program is the third one.

### First Evaluation for Best Move

One way to evaluate which move is best in position $R$, is to simply follow the child node with the most simulations. This is actually the most common way to go as explained here [4].

### Second Evaluation for Best Move

Another way to evaluate the best move, starting from the root of the MCTS tree, is to go with the child node that has the least wins, since those child nodes actually represent the wins of our opponent.

### Third Evaluation for Best Move

The final way we considered to evaluate the best move, was to take the node that gives the smallest winning ratio $\frac{V.w}{V.n}$, where $V$ is any child node of $R$, and $w$ and $n$ are its wins and simulations respectively. As explained before, since these ratios correspond to our opponent, we want the lowest one.

# Bibliography

[1] Wikipedia contributors, "Alpha–beta pruning — Wikipedia, the free encyclopedia," 2022. [Online; accessed 4-April-2022].

[2] O. Patashnik, "Qubic: 4× 4× 4 tic-tac-toe," *Mathematics Magazine*, vol. 53, no. 4, pp. 202–216, 1980.

[3] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2, pp. 235–256, 2002.

[4] Wikipedia contributors, "Monte carlo tree search — Wikipedia, the free encyclopedia," 2022. [Online; accessed 4-April-2022].