# AI 2 Course Project
# **3D Tic Tac Toe Game**

## Head

Dr. Parvin Razzaghi

## Teaching Assistant

Ashkan Rahmani Nejad

## Written By

Solmaz Mohamadi

Sepide Bahrami

IASBS – Winter 2020

# Game Understanding

**Tic Tac Toe** is a paper-and-pencil game for two players, *X* and *O*, who take turns marking the spaces in an n×n grid. The player who succeeds in placing four of their marks in a horizontal, vertical, or diagonal row is the winner. This project is a 3 dimensional version of Tic Tac Toe, which increases possible situations of winning up to 76 ones; and adds up to the game's challenges and complexity. We are going to provide a python program that plays this game.

# Data Structure, Initial State and Goal State

The **Data Structure** that has been used in our code consists of four 4*4 2D matrices. To output the game board in a readable way, we got some help from a similar code in GitHub, which will be cited, at the end.

- **Initial State:** can be any state given, including a three dimensional matrix. (Size is 4 by default).
- **Goal State:** is not a specific one since this is a game. The way that game ends depends on both players' decisions. Although the algorithm needs a measure to choose the best move by looking forward and estimating the value of a state in which it is in. What we have here is a score calculator for terminal states. There are 76 possible situations for scoring, and all of them are included. The terminal state is either a draw, or one of the players wins.

# Successor Function

Our **Successor** generates all possible states that can be reached from the current state of the game. So each time, depending on whose turn it is in the algorithm (which will be explained below), a new state is generated by changing only one of the cells (if they are empty). So the maximum number of generated states is when all of the cells are empty (consider initial state) which is 4*4*4 = 64.

# Game Algorithms:
# Minimax, Alpha Beta Pruning

**Utility**

Before developing the minimax algorithm, we had to define a utility function, which calculates the value of win, loss or game draw. We had covered all 76 possible ways to win the game with the help of loops in our win_or_loss() function. So if win, utility 10; if loss, utility -10; and if draw, utility zero is assigned to the value of that state.

**Minimax**

After defining our utility, we developed the minimax algorithm the same as the course book's given pseudocode. Minimax() function returns a new state, in which the computer 1 had chosen its best move and so the game will be continued by either human or computer 2's best move. In our trials, we did not get any answer at the very first moves of the game. However when we set the initial state to just have 4 possible moves while debugging, minimax finally answered after some time.

**Alpha-Beta Pruning**

As it was mentioned, because the game was too big to traverse all possible depths of the game and so it was very time consuming, we had to think about pruning some parts of the game tree in case to reduce the runtime of the minimax algorithm. Therefore, we developed Alpha-Beta Pruning, again the same as the course book's pseudocode. However, unfortunately neither this time we did not get an answer at the very first moves of the game. The game tree was still very big and we had to think about alternatives.

# Depth Limited Minimax,
# Depth Limited Alpha Beta Pruning

**Heuristic**

Before going through the depth limited version of the algorithms we have to define a heuristic function. So what does a heuristic function do?

Because in depth limited, we are somewhat in the middle of the game tree and there is no terminal state, so there would be no utility value for our current state.

In this case, we have to estimate the value of the state in which the game is in; and as it is an estimation, the win or loss of the game depends on how good the heuristic is.
Our heuristic gets a state and computes the possibility of winning for both players. In other words it only checks the cells that are likely to get that player the winning chance.

**Depth Limited Minimax**
In the depth limited algorithms we do not traverse all possible depths of the game tree but just some first depths. Specifying the depth depends on our possible runtime and space. In this project, we traverse until the algorithm is in depth 3 of the game with the use of our heuristic function. After limiting the depth, this time minimax answered at the very first moves after almost 1 and a half minutes which is observable rather than the unlimited minimax.

**Depth Limited Alpha-Beta Pruning**
So we can think of getting a better runtime by pruning the parts we do not need in the game tree. The simple depth limited minimax answered at the last step, but if we do pruning well, it is rational that we can do better in choosing the best move. Therefore, After adding the Alpha and Beta variables to the depth limited minimax we got the answer in less than half a minute which is less than the limited minimax without Alpha-Beta.
Generally, we started with the very basic minimax which did not answer at all at first steps to the situation in which we get the answer very quickly at the same steps.

# Analysis

**1. Game between Computer 1 and Computer 2 (Start with Computer 1)**

- **Winner:** Computer 2
- **Successors:** 704768
- **Average: 43**

**2. Game between Computer and Random (Start with Computer)**

- **Winner:** Computer
- **Successors:** 218918
- **Average:** 50

**3. Game between Computer and Human (Start with Computer)**

- **Winner:** Computer
- **Successors:** 473402
- **Average:** 39

| | 1 | 2 | 3 |
|---|---|---|---|
| **Winner** | Computer 2 | Computer | Human |
| **Successors** | 704768 | 218918 | 473402 |
| **Average** | 43 | 50 | 39 |

# Citations


1. https://www.cs.rochester.edu/u/brown/242/assts/studprojs/ttt10.pdf
2. https://github.com/adamdevigili/3D-Tic-Tac-Toe