# CSINTSY MCO2: Tic-Tac-Toe

**Members:**
Jadie, Joshue Salvador A
Ponce, Andre Dominic H.

## Agent Goal:

Given a classic 3x3 grid for a game of Tic-Tac-Toe, the agent must be able to win against its opponent by strategically placing its corresponding pieces on the board while abiding by the rules of Tic-Tac-Toe itself: (1) only one player can place a piece on a certain tile, (2) the players will take turns placing their pieces, (3) three pieces on a straight line is necessary to win, and (4) if the the grid has been fully filled with pieces already - and there is no winner - the game ends in a tie.

## Problem:

From an initial configuration that is an empty 3x3 grid and the first move given either to the agent or to the opponent, the agent must pick an empty tile that best leads to the desired goal state (agent wins or draw) until it is reached, depending on the moves of the opponent per state.

## States and configurations:

Given a state-space graph of possible Tic-Tac-Toe board configurations based on the turns made by both the agent and the opponent, different functions were implemented per behavior level. Below is the transition table of all implemented functions used depending on the agent's behavior level:

| Name | Condition | Transition | Effect |
|---|---|---|---|
| **getMovesLeft()** | --- | Gets the number of empty tiles in the current grid state, which also indicates the depth of the current grid state in the state-space graph.<br><br>Using a counter,<br>if the current tile being assessed is empty -> increase counter by 1. | Returns the value of the counter which indicates the number of empty tiles in the current grid state or the depth of the current grid state in the state-space graph.<br><br>The agent will then determine if it can still analyze and occupy tiles from the returned value. |
| **pickAIMove()** | Based on chosen behavior level,<br>Level 0: Random number generator to occupy an empty tile.<br>Level 1: Using a hard-coded table (series of if-else conditions).<br>Level 2: Performing depth 1 search with a self-made evaluation function<br>Level 3: Using minimax and self-made evaluation function<br>Level 4: Using minimax and self-made evaluation function, with depth counter | For the agent's turn, chooses an empty tile to occupy based on the given current grid state.<br><br>For all levels except level 0, the current grid state will be analyzed first, evaluating all possible moves and getting the best move that will benefit the agent itself towards reaching the goal state.<br><br>Executes **evaluate()** and **evalFunction()** if level 2-4.<br><br>Executes **minimax()** if level 3-4. | Agent occupies an empty tile on the board, indicating the next grid state and turn for the player. |
| **evaluate()**<br>**(Level 2-4 only)** | --- | Checks if the current grid state is a goal state (player win, agent win, or draw). If not, proceed to evaluating the current grid state by executing **evalFunction().** | Returns $+\infty$ indicating that the agent has won, $-\infty$ if the player won. If no one won in the current grid state, returns the value given by **evalFunction()**. |

| | | | |
|---|---|---|---|
| **evalFunction()** **(Level 2-4 only)** | When the current grid state is not a goal state as given by **evaluate()**. | Analyzes the current grid state by using the self-made evaluation function, producing a score value that tells whether it is beneficial to the agent or not. | Returns a score value indicating if the current grid state is beneficial to the agent or not. |
| **minimax()** **(Level 3-4 only)** | --- | Performs a **depth-first** search from the current grid state, analyzing all possible moves with **evaluate()** and getting the best move that leads to the desired goal state (win for the agent or draw).<br><br>Uses the Minimax game theory algorithm. | Agent will occupy an empty tile that will lead to the desired goal state based on the evaluation of the Minimax algorithm. |

*Figure 1. Tic-Tac-Toe AI Transition Table*

**Process towards reaching the goal state:**
An instance of a Tic-Tac-Toe game in this implementation will always generate a random number first - if it results in an even number, the first move goes to the AI; else, to the opponent. Once a game ends, the first turn is transferred to the other player in the next game.

Depending on the behavior level, the agent would react differently to the states it finds itself in. For example at level 0, when the player is about to win, the agent on this level cannot recognize this because it does not have any blocking capabilities - which could potentially result in a loss for the agent. This changes at levels 1 - 4 - wherein the agent is already capable of blocking the player's moves. For level 1, a hard-coded table (series of conditions) is used to find the best move for the agent. This hard-coded table is very similar to a person's mindset on the game, checking first if the other player has a chance of winning in order to block it, else check first the corners and the center tile to occupy with. Level 2 uses the evaluation function to analyze the current grid state but only from the current possible moves. Both levels 1 and 2 have similar behavior and cannot ensure that the goal state reached will always be a win for the agent or a draw as both only analyze the current possible moves, not checking the next succeeding states which might lead to a goal state wherein the player wins. Levels 3 and 4 are able to execute this by using the Minimax algorithm, checking each possible move as well as the next succeeding states from that current move taken.

For the evaluation function, we utilized the following self-made function:

$$f(n) = (P_{AI} + 2C_{AI} + O_{AI}) - (3A_{OP} + P_{OP})$$

Where $\boldsymbol{P_{AI}}$ is the number of possible wins for the agent, $\boldsymbol{C_{AI}}$ is the number of the agent's pieces found in corners, $\boldsymbol{O_{AI}}$ is the number of the agent's pieces in pairs found in opposite corners, $\boldsymbol{A_{OP}}$ is the number of opponent pieces in pairs that can win, and $\boldsymbol{P_{OP}}$ is the number of possible wins for the opponent.

The rationale behind this customized evaluation function is to enable the agent to perform strategies that will provide it with a state wherein no matter what move the opponent chooses, the agent will always win. Some example states where such strategies are applied are shown below.

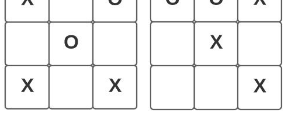

*Figure 2. Sample States with Guaranteed Win*

**Levels / Behaviors:**

*Level 0 (Random):*

This level does not implement any form of strategy at all. Using a random number generator, randomized row and column values are generated. If the values generated are not valid, it will generate a new set of values - this occurs in cases when the specific tile represented by the row and column values is occupied already. If it is valid, then the agent will place its piece on the tile found based on the randomly generated row and column values.

*Level 1 (Hard-coded):*

This level implements a shallow level of strategy. Based on a sequence of hard-coded conditions, the agent operating on this level will have the following priorities: (1) corner tiles, (2) center tile, and then (3) the remaining tiles. Along with this, this level is also capable of blocking the moves of its opponent - to prevent them from winning. This is again based on a series of hard-coded conditions.

*Level 2 (Depth 1 search with self-made evaluation function):*

This level implements a strategy that utilizes searching one depth lower than the current. This is the first level that utilizes the evaluation function discussed above, but this does not use the Minimax algorithm just yet. With the use of the evaluation function, the agent operating on this level is now capable of arriving at a state wherein it will surely win (see Figure 2). Even though this level already utilizes an evaluation function, just like the previous two levels, this can still be defeated by a human opponent - as it does not check the next succeeding states from that move taken which could possibly lead to a state wherein the player might win.

*Level 3 (Search with Minimax and self-made evaluation function):*

This level now uses the Minimax algorithm together with the evaluation function. Unlike level 2, the next succeeding states of each possible move is analyzed until the goal states have been reached. Using the evaluation function, the possible move that has the greatest positive score will be taken by the agent. It is always ensured that only goal states wherein the agent wins or a draw will be reached.

*Level 4 (Search with Minimax, depth counter, and self-made evaluation function):*

Similar to level 3 but uses a depth counter to determine the true best state. As some two states may both lead to the goal state wherein the agent wins therefore producing the same score ($+\infty$), the number of recursive searches will then be checked. If the move taken required a lesser number of searches than the other move then that move shall be taken instead. It indicates that the move is closer to the goal state based on the state-space graph.

**References:**

Aradhya, A. L. (2019). Minimax Algorithm in Game Theory. Retrieved December 26, 2020, from https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/

Russell, S. J., & Norvig, P. (2010). *Artificial intelligence: A Modern Approach* (3rd ed.). Upper Saddle River, USA: Prentice-Hall. doi:10.1016/j.artint.2011.01.005.