

**EAI 320: Artificial Intelligence**

Imtiaz Mukadam

U13083113

Assignment 4: Adversarial search

# Introduction

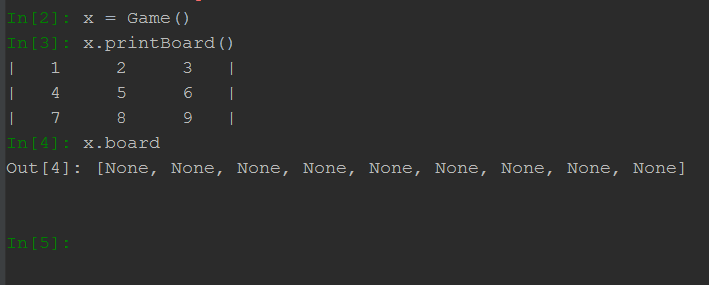
The Adversarial search strategies form the basis of AI in the game making world of computer science. These searches, as the name suggests, are for game scenarios that consists of 2 (or more) players that are considered adversaries. These search algorithms implement search strategies that aim to effectively reduce the odds of the opponent winning and find paths that increase and ultimately win the game for the player.

In this practical, we are tasked with implementing the Alpha-Beta algorithm generically that can be used to optimize moves for any game that follows its parametrization. The Alpha-Beta algorithm without pruning makes use of the ‘minimax’ function. The minimax function is used to obtain an integer value score of the current state with respect to which player has the current move. Players are defined as either ‘MIN’ or ‘MAX’, where lower scores are desirable to the MIN player and higher scores are desirable to the MAX player.

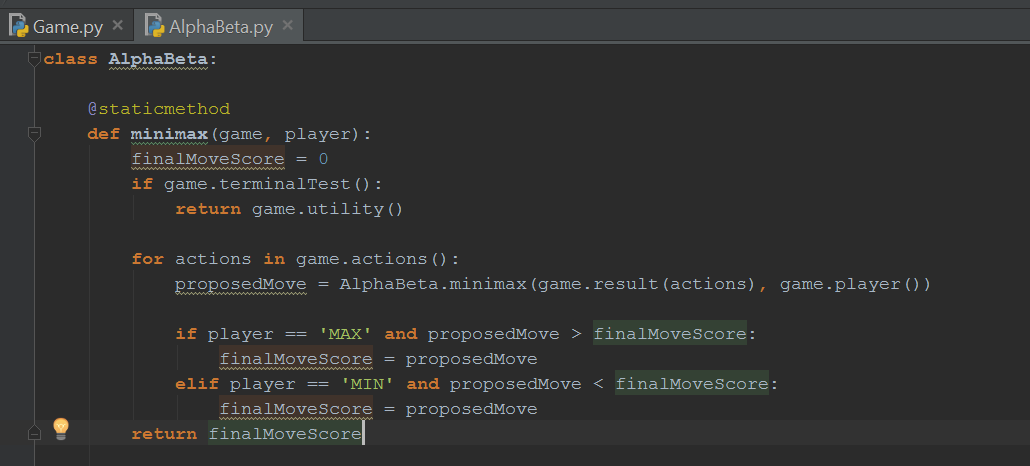
In addition to implementing the Alpha-Beta algorithm we are tasked with implementing a tic-tac-toe game itself, which makes use of interface functions player(), actions(), terminalTest(), and utility().

# Methodology

The tic-tac-toe game is implemented in the class named Game. This class constitutes the methods involved with managing a single run of a tic-tac-toe game. Once a game is started using the startGame() method, class functions manage the game by allocating moves and constantly checking if the game is over after each move. The following are the class members and main class functions that describe the mechanics of how the game is programed.

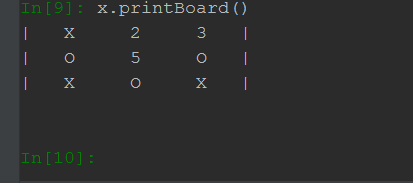
* \_\_init\_\_() (class members):
  + currentPlayer : the game keeps track of its own current player that updates and switches after each move is made with the call to the move() function.
  + board: this can be described as the current state of the game. A simple numerical array that describes each of the 9 playable positions on the board.
* actions(): returns an integer list of available moves to the current player.
* move(position): makes a move at the position specified on the board with the current players symbol and then calls turn()
* turn(): switches the currentPlayer variable to the opponent to signify the end of turn
* isGameOver(): returns true if the game is over, and false otherwise. This is done by checking if a winner is found or a draw occurred.
* findWinner(): peruses the board to find a winning condition and returns the wining players’ symbol. The function returns a None otherwise.
* startGame(): begins a run of the game in the command line, which prints the board and prompts the user for input in one of the 9 board locations. If a winner or draw occurs, the game ends and an output of the results are printed.
* terminalTest(),utility(),player(),actions(): interface functions that is accessed by the alpha-beta algorithm. Note that this version of the game returns ‘MIN’ for player X and ‘MAX’ for player O. This can easily be swapped around with relevant changes to the utility() function.

The alpha-beta function is implemented in a separate file, and is realized as follows:

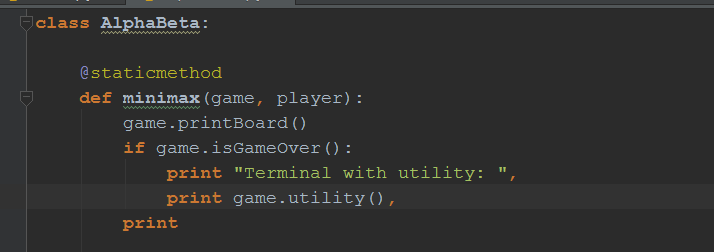


# Results

For the game state specified as:



By modifying the alpha-beta algorithm to print out each state it evaluates, we are able to generate an output of the nodes visited by the algorithm in the exact order, before it returns its final utility value. Code added to the beginning of the alpha-beta function:



Output generated when passing the given game state:

