# LAB 8 Adversarial Search – I

In this lab we consider two-player zero-sum games. Here a player only wins when another player loses. This can be modeled as where there is a single utility which one agent (the maximizing agent) is trying minimize and the other agent (the minimizing agent) is trying to minimize.

A game can be formally defined as a kind of search problem with the following elements:

- S0: The **initial state**, which specifies how the game is set up at the start.
- PLAYER(s): Defines which player has the move in a state.
- ACTIONS(s): Returns the set of legal moves in a state.
- RESULT(s, a): The **transition model**, which defines the result of a move.
- TERMINAL-TEST(s): A **terminal test**, which is true when the game is over and false otherwise. States where the game has ended are called **terminal states**.
- UTILITY(s, p): A **utility function** (also called an objective function or payoff function), defines the final numeric value for a game that ends in terminal state s for a player p. A **zero-sum game** is (confusingly) defined as one where the total payoff to all players is the same for every instance of the game.

## 1.1 Minimax Algorithm:

The **minimax algorithm** computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations. The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are **backed up** through the tree as the recursion unwinds. The minimax algorithm performs a complete depth-first exploration of the game tree.

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

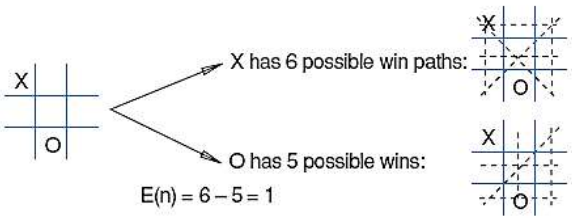
### 1.1.1 TIC-TAC-TOE Game:

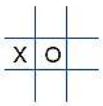
A Tic-Tac-Toe is a 2 player game, who take turns marking the spaces in a 3x3 grid. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row wins the game. The game is to be played between two people (in this program between HUMAN and COMPUTER). One of the player chooses 'O' and the other 'X' to mark their respective cells. The game starts with one of the players and the game ends when one of the players has one whole row/ column/ diagonal filled with his/her respective character ('O' or 'X'). If no one wins, then the game is said to be draw. In our program the moves taken by the computer and the human are chosen randomly. We use rand () function for this.

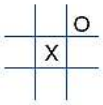
| | | |
|---|---|---|
| O | X | O |
| O | X | X |
| X | O | X |

### 1.1.2 Bounded Look ahead

Minimax, as we have defined it, is a very simple algorithm and is unsuitable for use in many games where the game tree is extremely large. The problem is that in order to run Minimax, the entire game tree must be examined, this is not possible due to the potential depth of the tree and the large branching factor. In such cases, **bounded look ahead** is very commonly used and can be combined with Minimax. The idea of bounded look ahead is that the search tree is only examined to a particular depth. All nodes at this depth are considered to be leaf nodes and are evaluated using a static evaluation function. In other words, the suggestion is to alter minimax in two ways: replace the utility function by a heuristic evaluation function EVAL, which estimates the position's utility, and replace the terminal test by a **cutoff test** that decides when to apply EVAL.




Heuristic is E(n) = M(n) – O(n)
where M(n) is the total of My possible winning lines
O(n) is total of Opponent's possible winning lines
E(n) is the total Evaluation for state n

## 1.2 Lab Tasks

**Exercise 8.1. Tic-Tac-Toe**

Formulate the problem.

**Exercise 8.2. Tic-Tac-Toe**

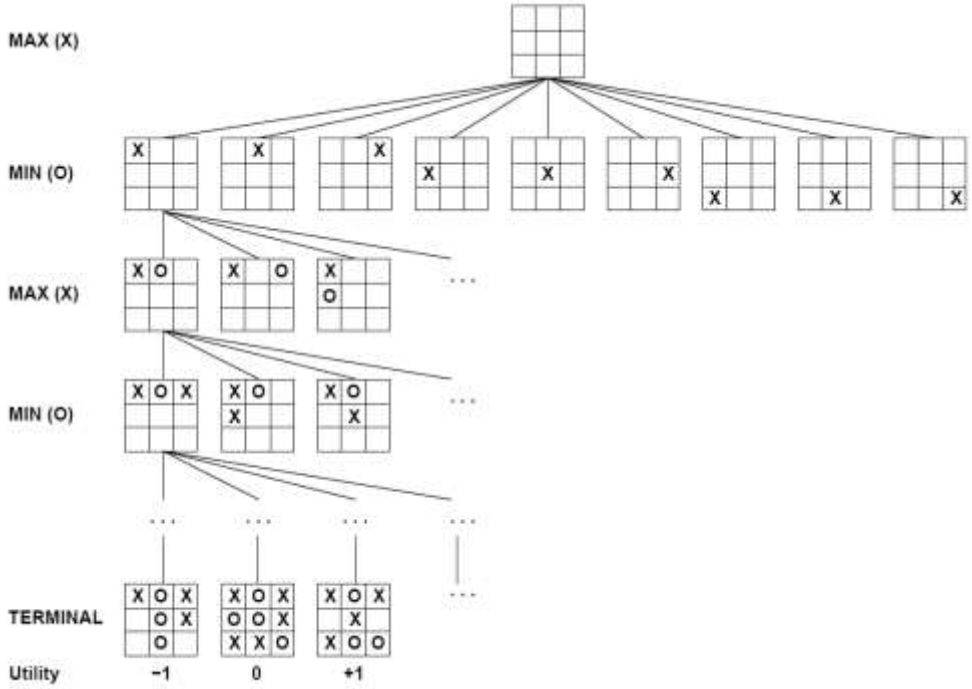
Implement the Minimax algorithm to rank the score of each possible move that an agent can do.

**Exercise 8.3. Tic-Tac-Toe**

Implement Minimax algorithm to solve the game. It should have a look-ahead of at least 3 half moves (ply's). In case of multiple moves giving same score a random choice should be made.

**Exercise 8.4. Tic-Tac-Toe**

You should make your agent more intelligent by employing a heuristic value that rewards each scenario. For this evaluate the relevant heuristic and implement the minimax algorithm with defined heuristic values.


MAX (X)
MIN (O)
MAX (X)
MIN (O)
TERMINAL
Utility
−1
0
+1