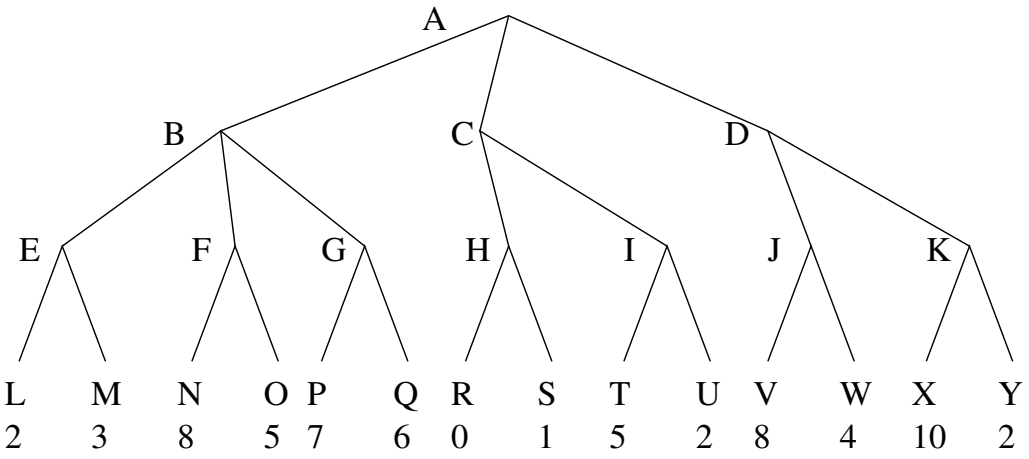
Lab 3: TicTacToe

Objectives:

- **To test your understanding of the minimax algorithm**
- **To practice working in a Python framework that is very similar to the one used in Assignment 1**
- **To implement a TicTacToe playing agent**

Adversarial search concepts

1. Describe the **Minimax** algorithm for search in two-player games.
2. Consider the following game tree in which the static scores (at the tip nodes) are all from the first player's point of view. Assume that the first player is the maximising player (i.e. MAX), and that high numbers represent better scores for MAX.



Use Minimax to determine which move the first player should choose.

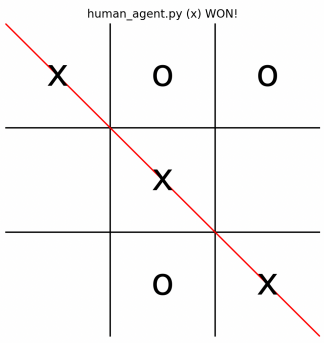
Adversarial search

Exercise 1: Implement the Minimax algorithm to play the game of Tic-Tac-Toe.

Download `cosc343TicTacToe.zip` from Blackboard and unzip it. Start VSCode, select `File->Open Folder...` and point it to the unzipped folder. Don't forget to set the interpreter to the cosc343 environment.

The files in `cosc343TicTacToe.zip` provide a similar framework to the one you will work with in your Assignment 1 (except, in this case, the framework is for an agent playing the game of TicTacToe). There is `cosc343TicTacToe.py` that runs the game (environment) and provides some helper functions, `settings.py` that allows you to configure which agents are playing the game, and also two implemented agents – `random_agent.py` and `human_agent.py`.

The default configuration is `settings.py` sets `random_agent.py` as player 1 and `human_agent.py` as player 2. When you run `cosc343TicTacToe.py` under this configuration, a Matplotlib[1] figure should come up and player 1 (random agent in this case) will make the first move as 'O'. Then player 2 (human agent – you) makes the second move. Just press the square where you want to make your mark and an 'X' should appear there. Then, the game will continue, with player 1 making random move and the game waiting for you to make the player 2 move. Keep playing until someone wins (hopefully not `random_agent.py` or TicTacToe is just not your game), or there is a draw.



Once the game finishes, close the figure. A second game will start, this time allowing player 2 to make the first move – so it will wait for you to click on a square where the first 'O' mark in this game is to be placed. Then player 1 gets a turn, and so on. After the second game `cosc343TicTacToe.py` terminates. If you want to stop the game beforehand, press the Stop button on PyCharm to kill the script.

The file `random_agent.py` provides a boilerplate for your agent. Make a copy of it calling the new file `my_agent.py`. Then, in `settings.py` change the "player1File" setting to `my_agent.py`. If you leave "player2File" set to `human_agent.py` you can can play against `my_agent.py`, though later you might change this setting to test `my_agent.py` against random agent... or even itself.

Take out the code in `my_agent.py` that chooses a random move and implementing minimax algorithm in its place. The boilerplate for `my_agent.py` should look something like this:

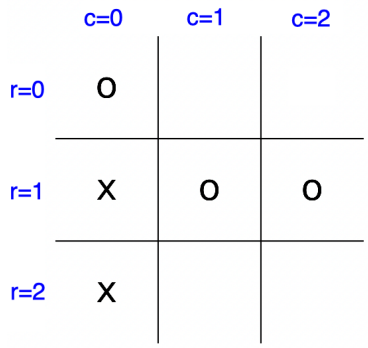
[1] Matplotlib is a Python library for creating plots.

```
class TicTacToeAgent():
    """
    A class that encapsulates the code dictating the
    behaviour of the TicTacToe playing agent

    Methods
    -------
    AgentFunction(percepts)
        Returns the move made by the agent given state
                of the game in percepts
    """

    def __init__(self, h):
        """Initialises the agent

        :param h: Handle to the figures showing the state of the board
        -- only used for human_agent.py to enable selecting the next
         move by clicking on the matplotlib figure.
        """
        pass

    def AgentFunction(self, percepts):
        """The agent function of the TicTacToe agent -- returns action
         relating the row and column of where to make the next move

        :param percepts: the state of the board a list of rows, each
        containing a value of three columns, where 0 identifies
        the empty square, 1 is a square with this agent's mark
        and -1 is a square with opponent's mark
        :return: tuple (r,c) where r is the row and c is the column
        index where this agent wants to place its mark
        """

        """
        Your minimax code here
        """
        return (r,c)
```

The code above implements a `TicTacToeAgent` class, which is instantiated in the `cosc343TicTacToe.py` script when the game is run. You probably can leave the constructor `__init__(self,h)` untouched – it's only needed for `human_agent.py` (but all agents need to implement the same initialiser signature, hence why it's there

containing single line, `pass`, telling the Python interpreter to do nothing). However, if you have some code that needs initialising (once) on instantiation of your agent, that's the method where you can put it.

Your implementation of the minimax algorithm should go into (or be invoked from) the `AgentFunction` method, which is called by `TicTacToe` whenever this agent needs to make its move. The `percepts` passed into the `AgentFucntion` is a double list (equivalent to a 2-dimensional array) relating the current state of the TicTacToe board. The size of the board is 3x3. The call to `percepts[r]`, where `r` is an integer such that $0 \leq$ `r` < 3, gives the r^{th} row of the board (which is a list of three integers providing the state of the board in that row). The call to `percepts[r][c]`, where `c` is also an integer $0 \leq$ `c` < 3, gives the state of the square at r^{th} row and c^{th} column. The state is encoded as follows: -1 is the opponent's (min's) mark, 0 is empty, 1 is this agent's (max's) mark. The game engine while player is 'X', which 'O' – the agent only needs to know that 1 is its mark, -1 is its opponent's.



The diagram above shows the relation of the `r` and `c` indexing to the state of the board. For the state in the image above, if the agent was playing 'O's, the `percepts` in its `AgentFucntion` would be [[1,0,0],[-1,1,1],[-1,0,0]]. And if the agent would be playing 'X's, the `percepts` would be [[-1,0,0],[1,-1,-1],[1,0,0]].

The action returned by `AgentFucntion` needs to be a tuple (r, c) indicating the row and column index where it wants to place its mark for its next move. That action must place a mark in an empty square (i.e. `percepts[r][c]` must be 0) otherwise, the game will crash.

To make the job of implementing the minimax algorithm a touch easier, some (hopefully useful) functions are provided for you in `cosc343TicTacToe.py`. The example syntax below assumes that your script imports helper like so:

```
from cosc343TicTacToe import maxs_possible_moves,\
                             mins_possible_moves,\
                             terminal, evaluate,\
```

```
                                    state_change_to_action,\
                                    remove_symmetries
```

To generate a list of board states with all possible moves made by max from the current `state`, you can do:

```
new_states = maxs_possible_moves(state)
```

Similarly, to generate a list of board states with all possible moves made by min from the current `state`, you can do:

```
new_states = mins_possible_moves(state)
```

The `new_states` will be a list of double lists representing 3x3 board states, each with additional 1 (when looking for max's moves) or -1 (when looking for min's moves). If you want to iterate over all the boards from `new_states` you can do:

```
for s in new_states:
  #Your code here, s is a 3x3 double list
  #...
```

In the example above, in each iteration of the loop, `s` will be a 3x3 double list (in other words, a list of 3 lists of 3 integers each), where the first index selects the row of the TicTacToe board and the second its column.

To evaluate the value of the board state `s` you can do:

```
v = evaluate(s)
```

When s is a terminal state, this value is either a 100 (indicating max's win), -100 (indicating min's win), or 0 for a draw. If s is not a terminal state, then evaluation function returns $I_{\text{MAX}} - I_{\text{MIN}}$, where I_{MAX} is the number of possible winning positions left on the board for the max player and I_{MIN} is the number of possible winning positions left on the board for the min player. If it's not immediately obvious to you how to use this in your minimax algorithm, it might be a good idea to go back and rewatch/reread notes from Lecture 5.

To check whether a given board state `s` is a terminal state (that is, either max or min have already won, or there are no more moves to make), you can use the `terminal(s)` method – it returns a boolean value of `True` when `s` is terminal, and `False` otherwise.

Finally, since the agent needs to return action in terms of a row and column index for its next move, you might need a function that translates the difference between the current `state` of the board and the `new_state` (presumed to contain one extra non-zero mark) using the following function:

```
r,c = state_change_to_action(state, new_state):
```

This should be everything you need to implement your minimax search without worrying about the particulars of the visualisation and the mechanics of playing the game against a human opponent. Make sure to implement the depth limited version of minimax...otherwise it might take a while (even in such a small game) to search through all the possibilities.

Test how well the agent plays, and how long it takes for it to make a move, when you increase the depth of the minimax tree.

One more thing that might come in handy – as discussed in the lecture, there are many symmetries in the TicTacToe board. If you want to speed up the performance of your algorithm, you can use a provided method to remove corresponding states of different rotational or reflection symmetry from the list of possible states to consider like so:

```
unique_states = remove_symmetries(new_states)
```

In the example above `new_states` is a list of states and `unique_states` is a selection of states from `new_states` list with unique symmetries.

Hints:

- `some_list = []` creates an empty list called `some_list`.
- `some_list.append(item)` adds an item to `some_list`.
- To find the index of the maximum or minimum item on a list (if you have, say, a list of numbers called `values`) use the Numpy[2] library like so:

```
import numpy as np

# To find the index of the minimum value on the list
I = np.argmin(values)

# To find the index of the maximum value on the list
I = np.argmax(values)
```

- Think recursion!

[2]Numpy is a Python library with numerous useful numerical methods, including data structures for working with vectors and matrices