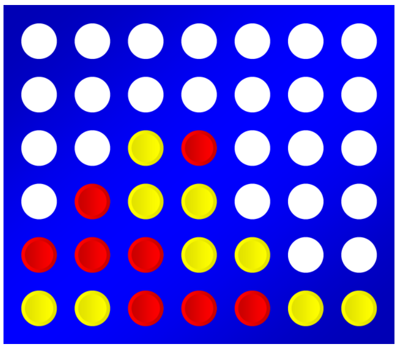
Intelligent Systems 1
Week 5 Lab Sheet

This week we are going to be playing Connect 4.



Connect 4 is a two player, zero-sum, symmetrical connection game, in which players take turns dropping one coloured disc from the top into a seven-column, six-row grid. The pieces fall straight down, occupying the lowest available space within the column. The objective of the game is to be the first to form a horizontal, vertical or diagonal line of four of one's own discs.

In order to represent Connect 4 programmatically, we need to define a game state. This has been provided for you in the form of the GameState interface. This interface is comprised of the following functions:

- **Clone**(): Creates a deep copy of the a game state.
- **GetMoves**(): Returns a list containing all the possible moves (actions) that an agent can execute at this state. A state with no available moves is also known as a terminal state, as the game cannot advance from this state onwards.
- **GetResult**(): To query the outcome of a game upon termination.
- **DoMove**(): The model of the environment. Performs an action in a given state, modifying state. This function is sometimes refered to as the **Step**() function.

I have also produced a piece of code which specifically implements the Connect 4 game. In connect4.py, you will see the **Connect4State** class and some helper functions, such as **PrintGameResults**() and **PlayGame**(). At the moment, the **PlayGame**() function plays randomly for both players.

1. Familiarise yourself with the code provided in connect4.py, make sure you know how the game is played by a computer (albeit randomly) before proceeding.

2. Implement a class which represents one node of a game tree.

    a. This class should be able to store all its child nodes, along with which player has just moved, the state of the game at that point, and the minimax value for that node.
    b. This class should also have functions which can update the minimax value according to the values of its children, and a function to generate its child nodes (i.e. the successor function).

3. Using your class from (2), implement minimax search for Connect 4. To start with, I suggest you stick with a small form of Connect 4, rather than the default 7x6 game. It may be useful to use the **Clone**() function so that you aren't playing directly on the game board.

    a. Implement a function which builds the entire game tree when passed an instance of **Connect4State**.

       N.B. using a Connect 4 game which is larger than about 3x4 will likely result in a game tree which is too big for your computer.

    b. Implement a function which looks like **PlayGame**() but which uses the minimax tree to determine the best move.

       N.B. using numpy's *argwhere* function and the *random.choice* function may be useful if you want to randomly select an index from a list.

    c. How well does your game work? How long does it take to generate the tree? How long to make a move? Why?

4. Implement a heuristic function that estimates the quality of a given non-terminal game state.
    a. Does this speed up your search?
    b. How does it affect play?

       N.B. using a heuristic allows you to limit the lookahead of the game AI, this should allow you to use larger game boards (I've found it works up to 10x10 on my laptop before it starts to struggle).

Challenge 1: Write a version of **PlayGame**() which allows the user to pick which move to make for one of the players. How well does your minimax agent play? What about your agent from (4)?

Challenge 2: Look up Monte Carlo Tree Search (MCTS) – can you extend your implementation to work with MCTS?