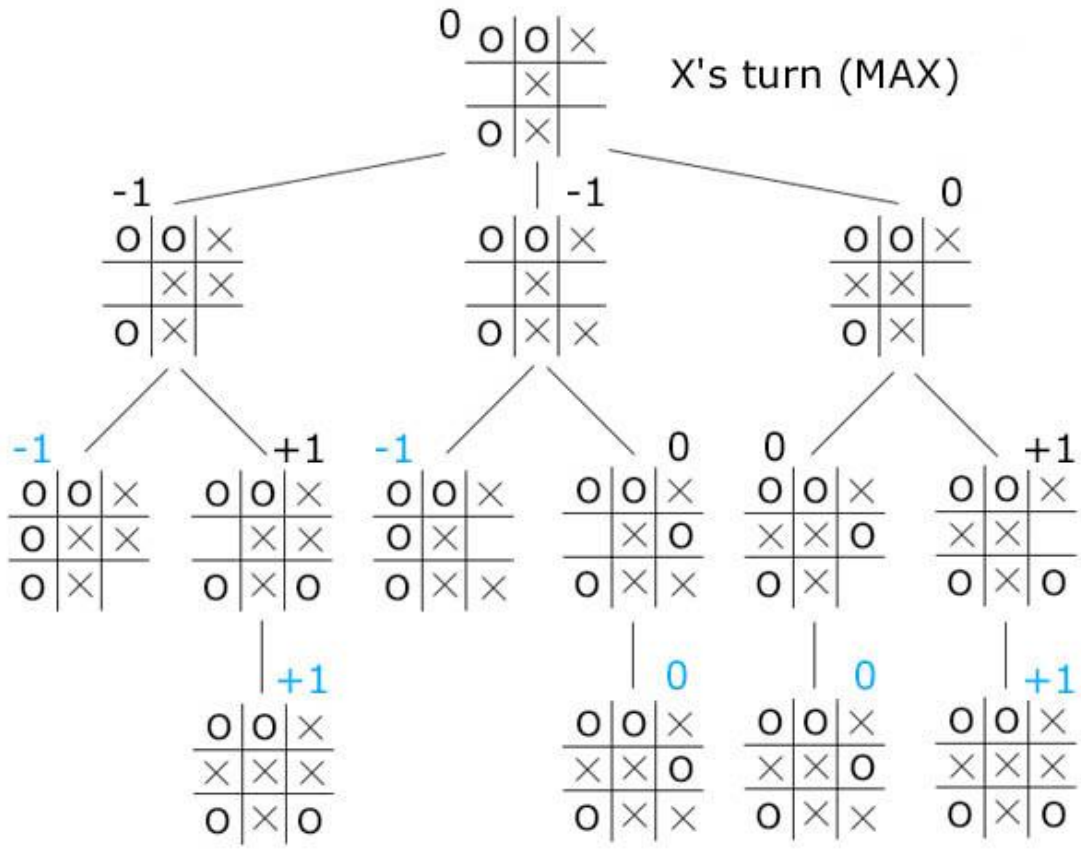
Tic-Tac-Toe Project

Any game between two players, where the latter take turns in playing their moves, can be described by a *game tree*. An example of a game tree for tic-tac-toe is the following:



This is the tic-tac-toe game tree when the initial state of the board being the one at the root of the tree (top); in case we are using a simple list representation of a game state, this is the list [O,O,X, ,X, ,O,X,]. Note that with this initial state (equal numbers of Xs and Os, X-player always plays first, this is not a terminal (winning or draw) state), we know that the first to move is the X-player. The next level of the tree shows the three new states that can result from X-player's move (there are three such possible moves, so that's why we have three alternatives). The next level of the tree shows all possible game states after the O-player moves, etc. Note the leaf states of the tree (those with a blue number next to them); these are all the possible terminal states this particular initial state can produce.

Moreover, by following the tree path starting at the root and finishing at a leaf/terminal state, we can also derive which move each player must play in order to finish the game in this particular leaf.

So far, the game tree encodes all possible plays (starting from a particular initial state) that can be played by the two players, and all possible states that can produced during the game, but doesn't yet encode *who's the winner*, i.e., we are missing an *evaluation* for each particular position. This is the role of the numbers you see next to each state. For tic-tac-toe, let's assume that the X-player tries to make the final result as positive as possible (i.e., she's a *MAXimizer*), and the O-player tries to make the final result as negative as possible (i.e., he's a *MINimizer*). Let's assume that the maximum possible (most positive) value is +1, and the minimum possible (most negative) value is –1. Then it's easy to do the evaluation of the terminal states (blue numbers in the figure): if the X-player wins at a terminal state, then the value for this state is +1, if the O-player wins, then the value is -1, other wise (draw) the value is 0. Similarly, the value of a game state that is an internal node of the tree is an estimate of the profit this

particular state can lead to: the more positive this value is, the more favourable for the MAX player this state is, and the opposite will hold for the MIN player.

Unfortunately, the leaves (terminal states) are the only states whose value we know *a* priori, i.e., from the state itself and without looking at the tree. For the rest of the states, we need to look ahead (i.e., deeper in the tree) for a few moves, and somehow use values we have calculated in deeper levels to calculate a value for the current state; after these values have been calculated, the player will move to the most positive or negative (depending on whether it's the MAX or MIN player that moves, respectively) of the available next states (i.e., in game-theoretical jargon, we assume *rational players*). For example, if the values calculated are the ones shown in the figure above, the best alternative for the X-player at the initial state is to make the rightmost move; of course, her best prospects have been calculated to be a draw (value=0), but this is still better than a final outcome of -1! The algorithm that calculates these values, therefore, is at the heart of the AI in agame. We will implement three such algorithms.

- **Minimax:** This is the algorithm that implements the value calculation outlined above. It goes over the whole game tree, calculating the values recursively: the base cases are obviously the leaves, and after calculating the values of all the children for a state, the value of this state is the maximum (minimum) children value, if in this state it is the MAX(MIN) player's turn to move. You can find the algorithm in pseudo-code in Yosen Lin's page:

https://www.ocf.berkeley.edu/~yosenl/extras/alphabeta/alphabeta.html

1. (10 points) Implement an algorithm `TreeBuild(S,player)` that, given an initial state `S` and `player`=X or O, returns a list of all game states generated by the game tree with root `S` and starting with `player` moving, each in the following format: *[game board, value, whose move, move]*. For example, for the root state in the figure above, the state description is `[O,O,X, ,X, ,O,X, ,0,X,(1,0)]`, where the first 9 elements of the list describe the board, then comes the value (0), then the player who is to make the next move (X-player), and finally the move (an X at (1,0)). Note that the coordinates of the upper-left corner is (0,0) and those of the lower-right (2,2). Write a program `testminimax.py` that, after asking the user for an initial `S` and `player`, runs `TreeBuild` and prints all the states in the game tree, one at a line as follows:

…
`State=[O,O,X, ,X, ,O,X, ], Value=0`
…

2. (10 points) Build a tic-tac-toe game `minimaxttt.py` that has three modes for the user to pick: 0-player, 1-player, 2-player. In the 0-player mode, the computer plays the role of both the X-player and the O-player, in the 1-player mode the computer plays the O-player, and in the 2-player mode both players are human. The user also picks an initial state for the game. Any time the computer has to play a move, it calls `TreeBuild(S,player)` to compute the value of the current state and the next move. Then it makes that move, and the game continues.

- **Alpha-beta pruning:** The exhaustive search of the whole game tree that the Minimax algorithm performs in order to calculate a position value and the next move becomes prohibitively slow when applied to a more complicated game like chess. In order to avoid this exhaustive search, we can try to avoid (i.e., *prune*) subtrees that we know for sure are not going to give a better value for the player at the current state. In alpha-beta pruning, at every node of the tree we maintain two values, *alpha* and *beta*. Alpha is the current estimate of the best possible move the player of the state can make. Beta is the current estimate of the best possible move the opponent can make. If at any time, alpha >= beta, then the opponent's best move can force a worse position than the player's best move so far, and so there is no need to further evaluate the current state, and we can ignore the whole subtree of the game tree rooted at this state. The

same pruning condition applies for both MAX and MIN cases: if the current state player is MAX, then we are looking for the move that yields the best alpha, while it is the MIN player, we are looking for the move that yields the best beta. A detailed description of the alpha-beta pruning algorithm can be found in Bruce Rosen's page:
http://web.cs.ucla.edu/~rosen/161/notes/alphabeta.html

and pseudocode for the algorithm can be found in Yosen Lin's page:
http://web.cs.ucla.edu/~rosen/161/notes/alphabeta.html

3. (10 points) Implement an algorithm `TreeBuildAB(S,player)` that works like `TreeBuild(S,player)` above, but implements the alpha-beta pruning algorithm, i.e., instead of a single *value* it now maintains two numbers, *alpha* and *beta*. Then implement `testab.py` that works exactly as `testminimax.py`, but uses `TreeBuildAB(S,player)` instead of `TreeBuild(S,player)`, and with output

```
…
State=[O,O,X, ,X, ,O,X, ], Value=0
…
```

4. (10 points) Build a tic-tac-toe game `abttt.py` that works exactly as `minimaxttt.py` but the computer uses alpha-beta pruning to pick its next move.