Lecture Worksheet
November 19, 2015

Solving Tic-Tac-Toe

When an optimal player plays tic-tac-toe, they can never lose. With a simple algorithm, we can make a tic-tac-toe game where you, the programmer, can never beat your own algorithm!

Representing the Board State

There are infinite ways to represent the current board ("the board state") in our algorithm. Arguably one of the simplest is to do it as a simple list of characters:

X	X	O
X	O	
O	O	X

→

[0]	[1]	[2]
[3]	[4]	[5]
[6]	[7]	[8]

In the above figure, the board would be:

```
["X", "X", "O", "X", "O", "-", "O", "O", "X"]
```

…this board state is a win for O, as they connected three Os along a diagonal.

Q: What are all possible indexes that must be checked to determine if a player has won?

Puzzle #1:

Complete the **`checkState`** function in Python. This function takes in one argument, **`board`** (a List, as defined above), and should return:

- 1, if X has won
- -1, if O has won
- 0, if no one has won

The following are example calls to **`checkState`** and the expected return values:

```
checkState(["X","X","O","X","O","-","O","O","X"]) → -1
checkState(["-","-","-","-","-","-","-","-","-"]) → 0
checkState(["X","X","X","O","O","-","-","-","-"]) → 1
checkState(["X","O","X","O","O","X","X","X","O"]) → 0
```

Traversing the Game States

To ensure that we can craft an algorithm that will never lose, we need to find all of the paths to victory and make sure to avoid all the paths to defeat!

To do that, we need to evaluate every possible game state. If it's currently Os turn to go on the following board:

X	X	O
	X	
O		

Current board state (as shown above):

```
["X","X","O","-","X","-","O","-","-"]
```

Next board states (after O makes a play):

```
["X","X","O","O","X","-","O","-","-"]
["X","X","O","-","X","O","O","-","-"]
["X","X","O","-","X","-","O","O","-"]
["X","X","O","-","X","-","O","-","O"]
```

Puzzle #2:

Complete the **`getNextStates`** function in Python. This function takes in two arguments, **`board`** (the current board, a List) and **`turn`** (the current player to go next, either "X" or "O").

The function should return a list of dictionaries of all the possible next board states that can occur after the turn is made. Each element in the dictionary should contain exactly two values:

- **`board`**, the future board state
- **`move`**, the index of the array that the move was made

In the example above, the returned data structure would be:

```
[
  {"board": ["X","X","O","O","X","-","O","-","-"], "move" 3},
  {"board": ["X","X","O","-","X","O","O","-","-"], "move" 5},
  {"board": ["X","X","O","-","X","-","O","O","-"], "move" 7},
  {"board": ["X","X","O","-","X","-","O","-","O"], "move" 8},
]
```

Finding the Winning Move

Finally, we can find the winning move! For every board state, we can always win by programming a winning algorithm:

- If O can win, make that move.
- If X can win, avoid the move that allowed X to win.
- If the game has not finished yet, find the future board states from the current board state.

We'll work on this one together!