**The lecture 11**

**A Perfect but Fatalist Player**

Implementing the above algorithm will get you to a point where your tic tac toe game can't be beat. But and interesting nuance that I discovered while testing is that a perfect player must always be perfect. In other words, in a situation where the perfect player eventually will lose or draw, the decisions on the next move are rather fatalistic. The algorithm essentially says: "hey I'm gonna lose anyway, so it really doesn't matter if I lose in the next more or 6 moves from now."

I discovered this by passing an obviously rigged board, or one with a "mistake" in it to the algorithm and asked for the next best move. I would have expected the perfect player to at least put up a fight and block my immediate win. It however, did not:



Let's see what is happening here by looking through the possible move tree (Note, I've removed some of the possible states for clarity):

* Given the board state 1 where both players are playing perfectly, and O is the computer player. O choses the move in state 5 and then immediately loses when X wins in state 9.
* But if O blocks X's win as in state 3, X will obviously block O's potential win as shown in state 7.
* This puts two certain wins for X as shown in state 10 and 11, so no matter which move O picks in state 7, X will ultimately win.

As a result of these scenarios, and the fact that we are iterating through each blank space, from left to right, top to bottom, all moves being equal, that is, resulting in a lose for O, the last move will be chosen as shown in state 5, as it is the last of the available moves in state 1. The array of moves being: [top-left, top-right, middle-left, middle-center].

What is a gosh-darn, tic tac toe master to do?



**Fighting the Good Fight: Depth**

The key improvement to this algorithm, such that, no matter the board arrangement, the perfect player will play perfectly unto its demise, is to take the "depth" or number of turns till the end of the game into account. Basically the perfect player should play perfectly, but prolong the game as much as possible.

In order to achieve this we will subtract the depth, that is the number of turns, or recursions, from the end game score, the more turns the lower the score, the fewer turns the higher the score. Updating our code from above we have something that looks like this:

def score(game, depth)

 if game.win?(@player)

 return 10 - depth

 elsif game.win?(@opponent)

 return depth - 10

 else

 return 0

 end

end

def minimax(game, depth)

 return score(game) if game.over?

 depth += 1

 scores = [] # an array of scores

 moves = [] # an array of moves

 # Populate the scores array, recursing as needed

 game.get\_available\_moves.each do |move|

 possible\_game = game.get\_new\_state(move)

 scores.push minimax(possible\_game, depth)

 moves.push move

 end

 # Do the min or the max calculation

 if game.active\_turn == @player

 # This is the max calculation

 max\_score\_index = scores.each\_with\_index.max[1]

 @choice = moves[max\_score\_index]

 return scores[max\_score\_index]

 else

 # This is the min calculation

 min\_score\_index = scores.each\_with\_index.min[1]

 @choice = moves[min\_score\_index]

 return scores[min\_score\_index]

 end

end

So each time we invoke minimax, depth is incremented by 1 and when the end game state is ultimately calculated, the score is adjusted by depth. Let's see how this looks in our move tree:



This time the depth (Shown in black on the left) causes the score to differ for each end state, and because the level 0 part of minimax will try to maximize the available scores (because O is the turn taking player), the -6 score will be chosen as it is greater than the other states with a score of -8. And so even faced with certain death, our trusty, perfect player now will choose the blocking move, rather than commit honor death.