Game Trees & Minimax

# Introduction to game trees

So to start out this topic we will be playing a game. In this game 2 players are in an intense stand off, plucking petals of a flower. You each take in turns, plucking either 1, 2 or 3 petals of each go. Whoever plucks the last flower off the flower loses.

## The Game

So lets play!

Game 1: the number of petals on the flower is 6. I will take the first go. (remove 1).

Let them take their go.

Remove the amount of petals it takes to get to 1.

Then you win.

Game 2: the number of petals on the flower is 13. Let them go first.

Now remove the number of petals to get to 9.

Their go.

Get to 5 petals now.

Their go.

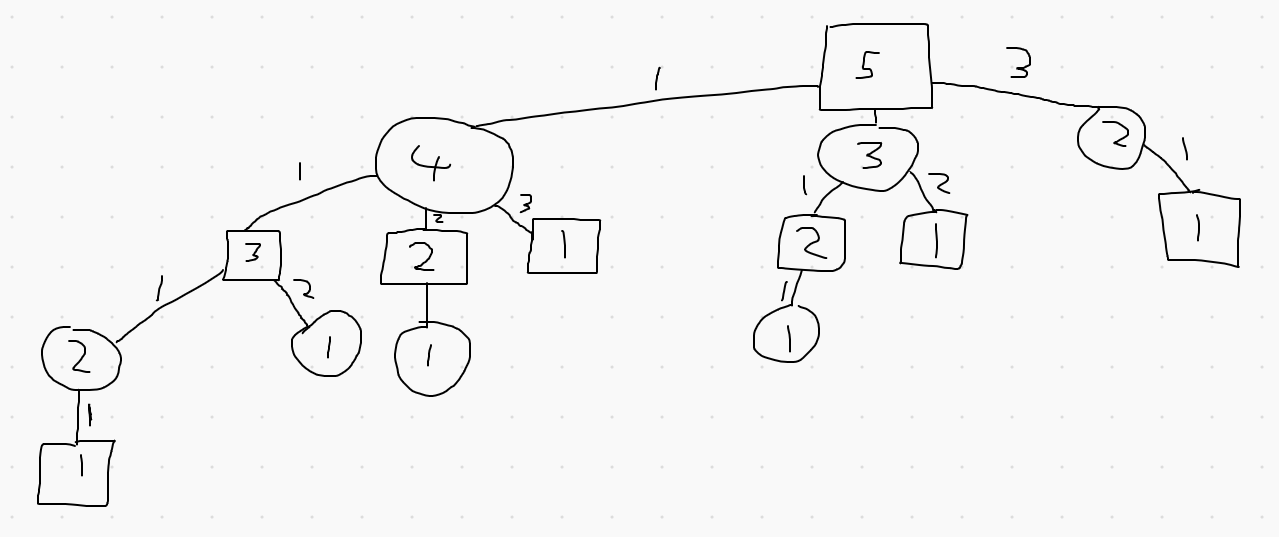
Get to 1 petal now.

Then you win.

What I have shown here is the power of game trees by always being able to win as I got to decide what number of petals to start. Using game trees we will be able to know whether a game is winnable if both players play **optimally**.

## Game trees

Game trees are a method of representing decisions being made in a game on a tree. For the game above here is the tree of when the petals = 5.

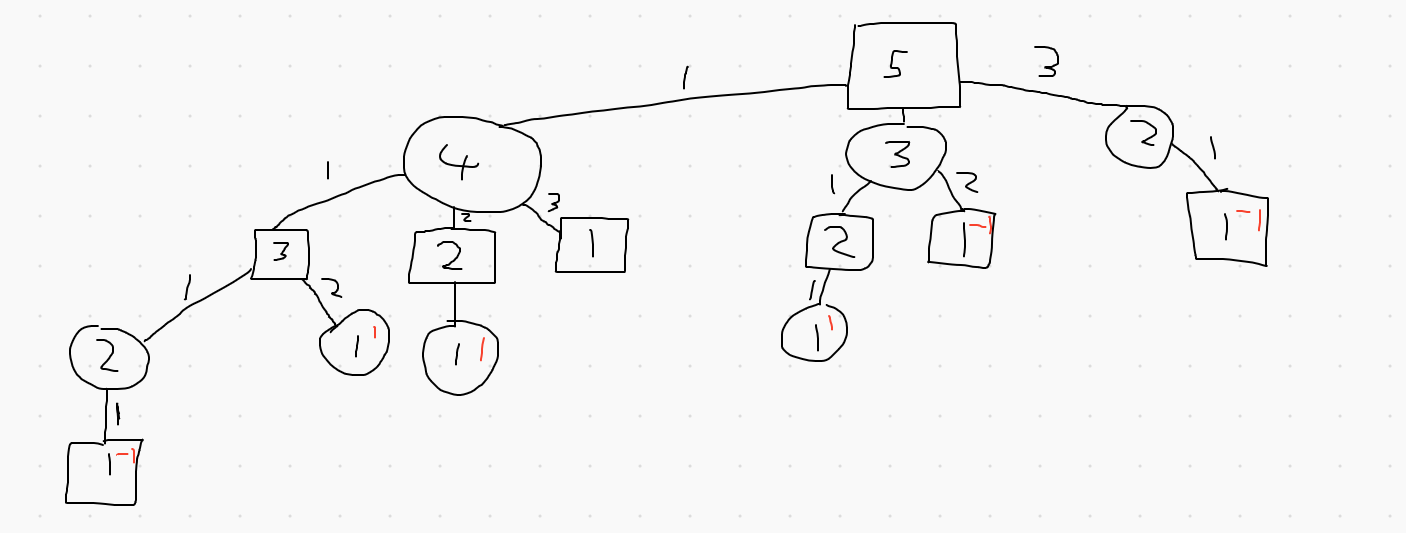


The squares represent player1’s go, and circles represent player2’s go. So any leaf node is a loosing state. The connections show the move they player took, so if the connection is labelled 1 it means the player in the parent node removed 1 petal. The number in the centre is the number of petals.

We will be looking at this game from the “perspective” of player 1, so we will be seeing if player 1 can win. For each node in the graph we want to calculate it .

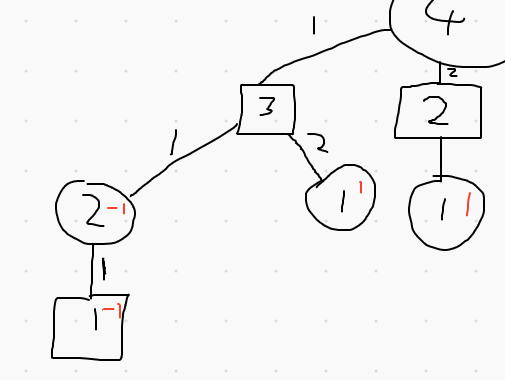
E(x) is going to show the likelihood of each player winning, where 1 is Player 1 will win and -1 being Player 2 will win. represents here the game state, in this case the entire game state can be represented by the number of petals left and who’s turn it is.

So we start at the leaf nodes and work our way up, as if you are at the leaf node you know who will win and who will lose ( is represented in red):

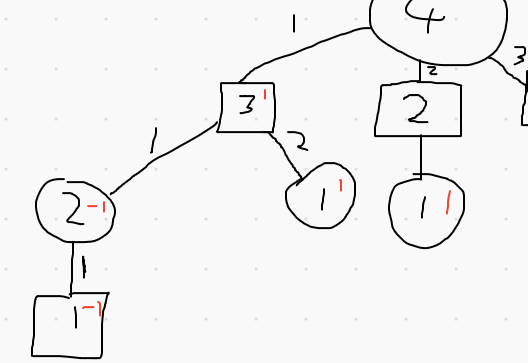


(I missed a leaf node in the child of 4)

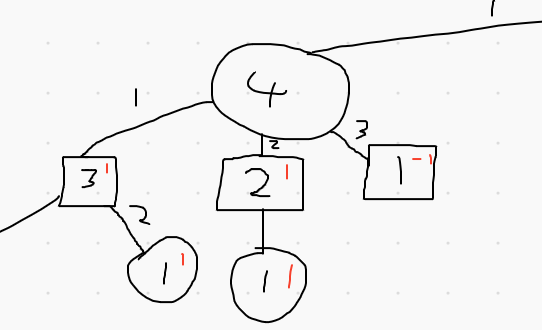
Now we will work our way up the tree. We assume here that both players will play optimally. So if the current node is a square (player 1) we will always choose the max of the child nodes. This is the opposite for a circle node (player 2) they will always choose the min of the child nodes. If there is only 1 child node then the parent node just takes on the children’s . So starting on the far left:



The value is inherited from the child. Then we reach our first decision, it is a square node, so we take the max ox the child nodes, so :

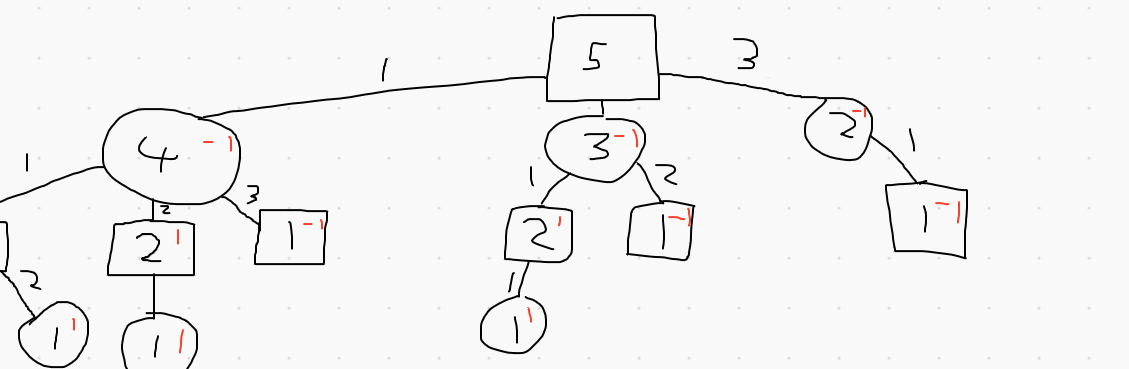


Skip forward a bit:



This is a circle node, so the minimum will be taken.

Skip more until you get to the root node:



So we’ve reached the root node, all options are -1, this means that if player 2 plays optimally then player 1 cannot win. So .

## Minimax

The procedure described above is called Minimax, Player 1 is trying to maximise the chance (E(x)) of them winning while Player 2 tries to minimise it. This procedure is very useful in determining what move to make next but lets say for the game above we have 100 petals, this is going to be a very large tree and take a very long time to compute. We will help with this by doing two key optimisations.

From now on I will be talking about an arbitrary game, not the game above.

## Estimating

For games such as chess you can only look a few moves in the future before the tree gets too complex for most computers, and you cannot get anywhere near to simulating a full game. This means you cannot get a or as you cannot be certain who will win without simulating the possibilities. So instead you can estimate using a function that take in the state of the game and gives out a value. For a game like chess you could have a function that give a weight to each piece type, so queen could be 100, rook 20 etc. Then add up all the white pieces and subtracts all the black pieces. Then do the algorithm as seen before, with player 1 (white) picking the max of the children and player 2 (black) picking the min of the nodes.

## Alpha-beta pruning

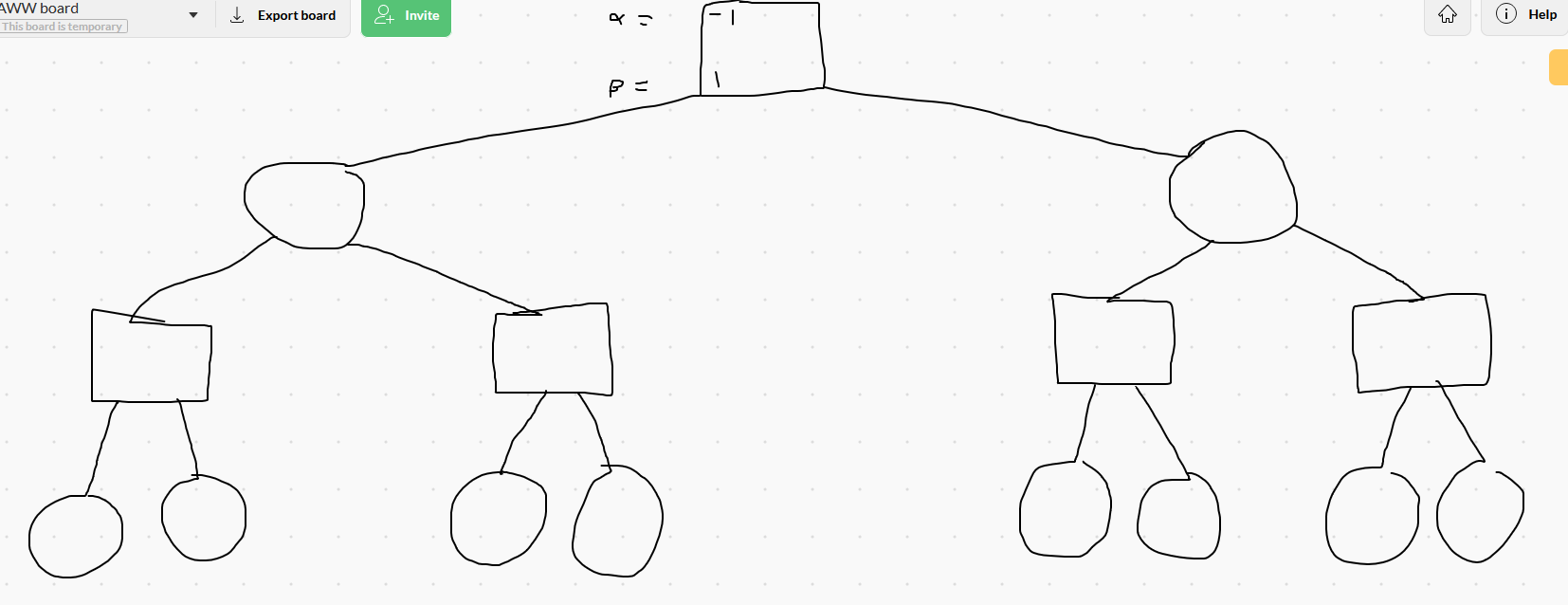
This is where things get a little more complicated. But to sum up this algorithm, it is a way of eliminating branches of the trees because the game would never end up in that state.

The basic algorithm for this is:

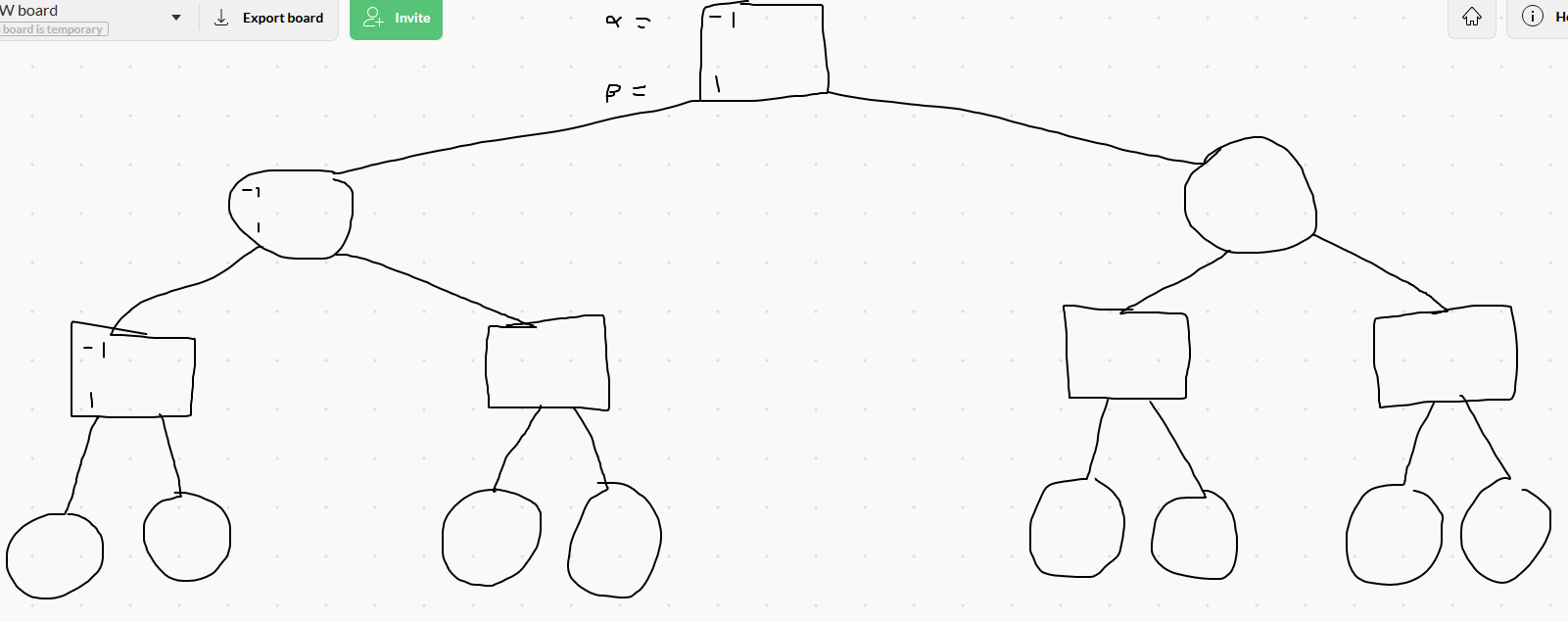


Okay so let’s carry this algorithm out on an arbitrary game to demonstrate how it works. For this example we will have a max depth of 3.

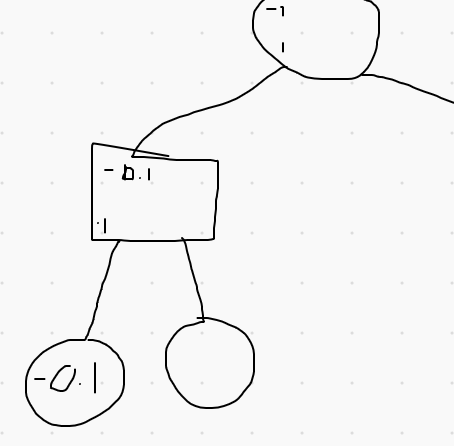
As a quick note, the value at the centre of the node no longer will represent the game state, it will show the for that node.



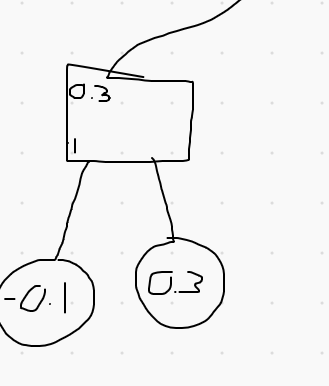
We have set the alpha, beta value of the root to -1 for alpha and 1 for beta. We start by going down the left side of the tree, passing on the alpha and beta value to each node we get to until we reach a leaf node:



Since we have reached a leaf node we need to estimate the , lets say it -0.1 (the choice to be -0.1 doesn’t mean anything it is just a random value).

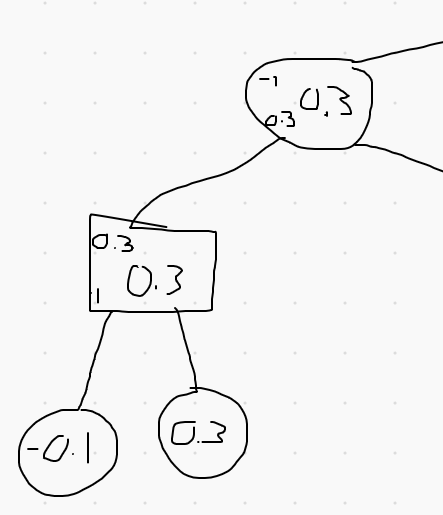


Alpha is now -0.1 as it is the best we have seen so far and we are player 1 so we are trying to find the max. alpha is still less than beta so we keep going to the next node.

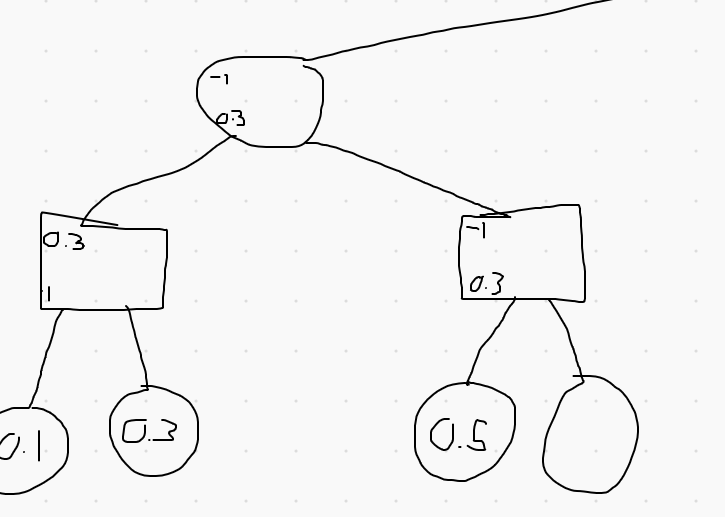


Alpha is now 0.3 as it is greater than -0.1. We have no children left to check so we return the child with the largest , so 0.3.

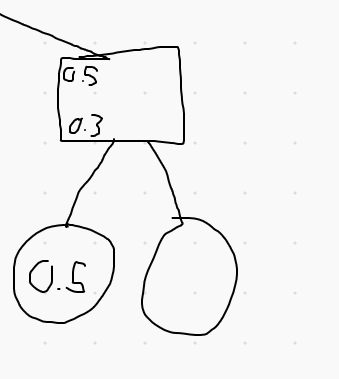
This now replaces the beta value in the parent node with 0.3.



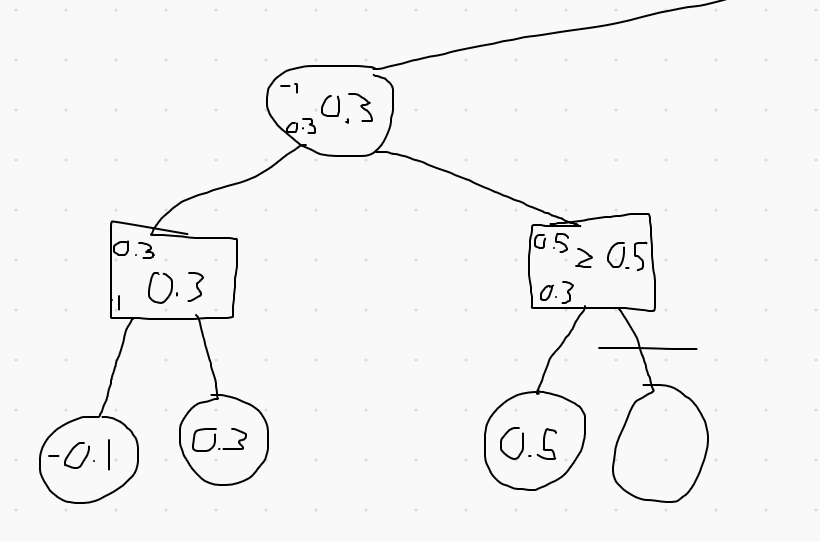
Now check the right branch:



The leaf node has a value of 0.5, we now replace the -1 with 0.5:

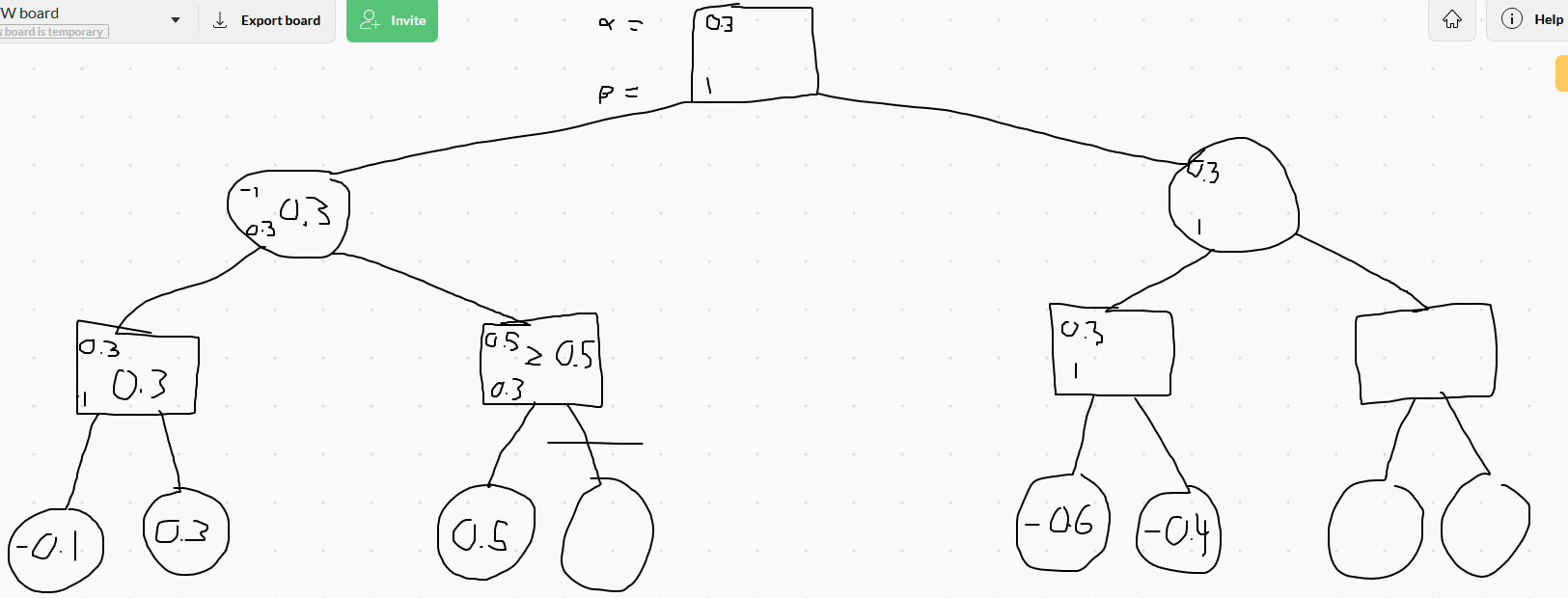


Now alpha is greater than beta, showing that player 2 would not pick this route just based off of the nodes we have searched, so we break out of the loop and return 0.5. This shows this route would give a value of at least 0.5:

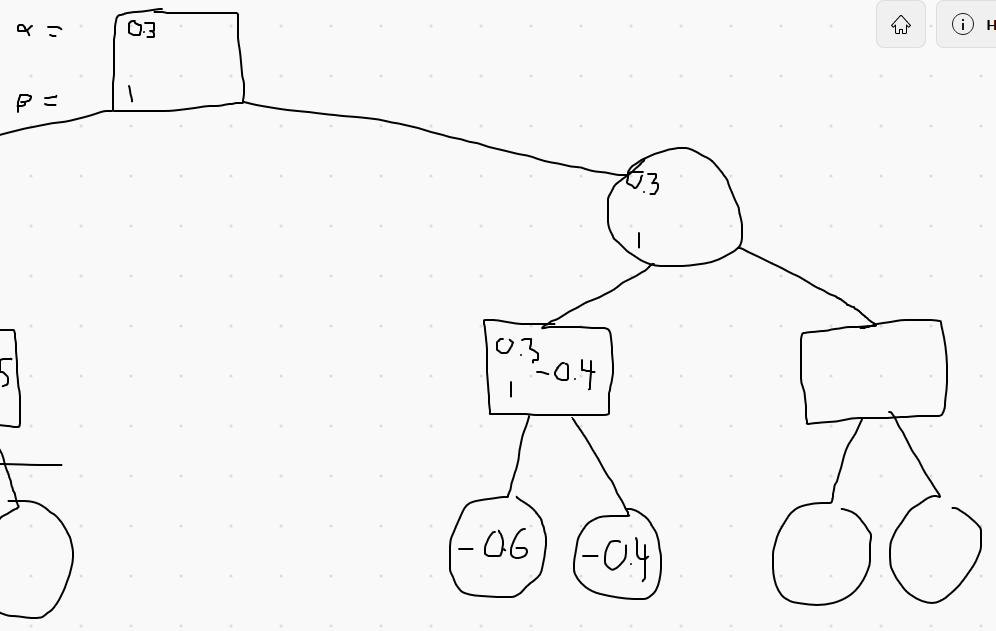


In this case alpha beta pruning has only saved us 1 recursion but it can do a lot more as you will soon see.

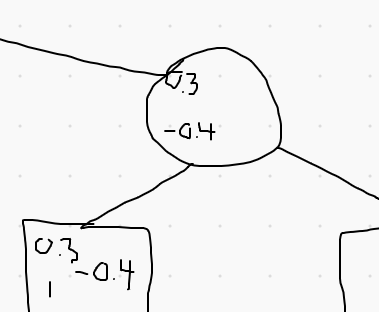
Now the root takes on 0.3 as alpha and then investigates the right hand root.



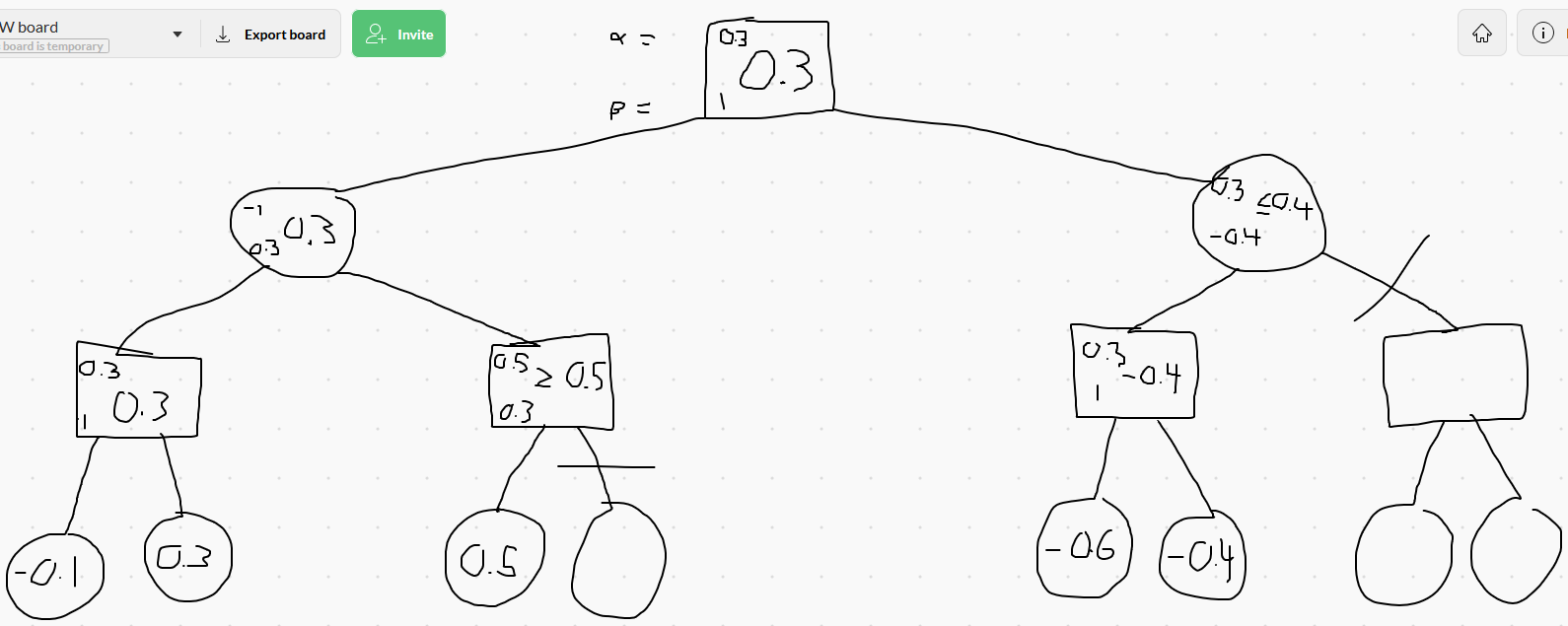
Now we look at the two children, since they are both less than 0.3, alpha does not change and the node returns -0.4 as it’s the max of the two children.



Now beta will become -0.4 as it’s less than 1.



Now this means that beta is less than alpha, showing us that this state will never be reached if the players play optimally. So this node returns -0.4.



We cut off the right most section of the tree because we know if the players play optimally, they will never go down that route. So overall the player in this game state knows that if they make the decision that leads down the left branch they have a higher chance of winning (according to your estimate for ).