TECHNICAL DOCUMENTATION

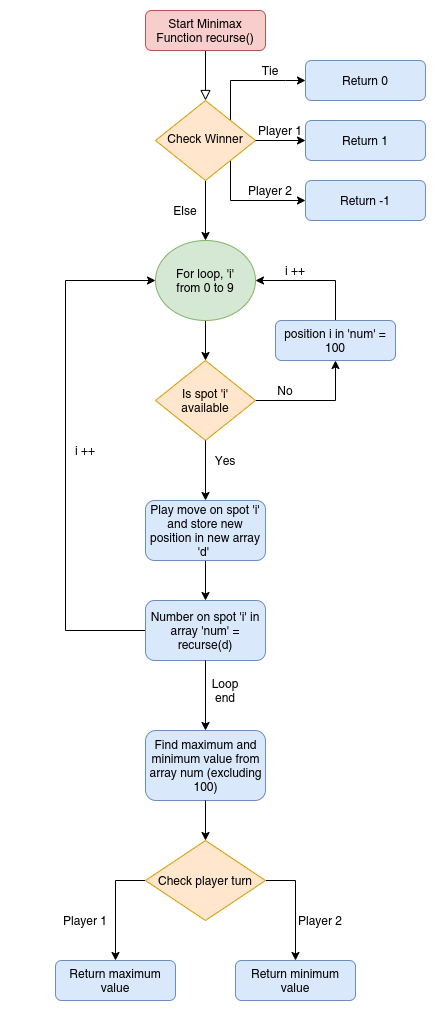
TIC-TAC-TOE

How the Program Works

Minimax Algorithm

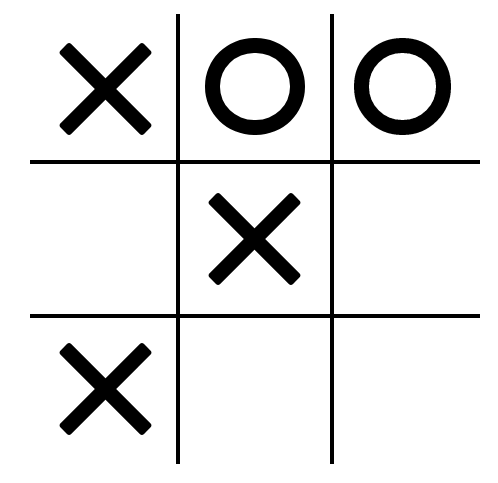
Simply put, Minimax is a kind of backtracking algorithm used in two player turn-by-turn games that solves a particular position by assuming that both players play perfectly. It plays all of the possible moves for both of the players and tries to minimize the maximum loss for the player. If a minimax algorithm is given infinite computing power, it can solve for any two player game like Chess, Go, Tic-Tac-Toe or Othello, and provide a perfect moveset for any position. However, games like chess or go have too many possible combinations to solve completely, so game engines for these games cannot realistically provide perfect moves. Tic-Tac-Toe, on the other hand, has less than 3^9 (=19683) combinations, which can be solved completely in little time.

In the written program, minimax algorithm is applied on a given position of a Tic-Tac-Toe ‘board’ by calling a function ‘recurse’ which has the given position to be solved as a parameter. It doesn't return the best possible move, but instead returns if the winner is going to be player 1, player 2 or if it is going to be a tie if both players play ideally. The function returns 1 , -1 or 0 respectively for these cases. The position of the board is represented by an integer array of length 9 (variable ‘b’) with 0 for available moves, 1 for player 1 moves (‘X’) and 2 for player 2 moves (‘O’). An array ‘num’ is also declared which stores 0,1,-1 for each position to see which player would win if those specific moves are played by the player. It stores ‘null’ (or 100 in my program) for any position which cannot be played because it is already occupied. The program works according to the following logic:



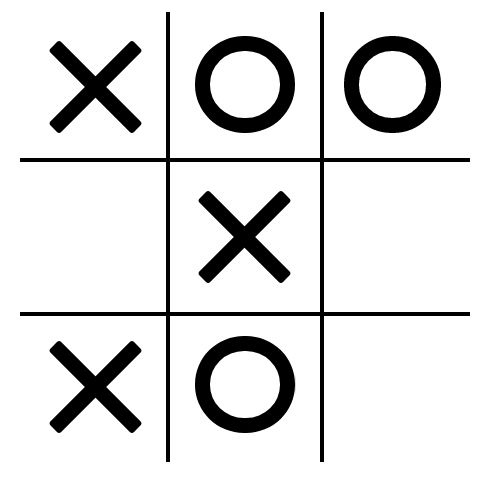
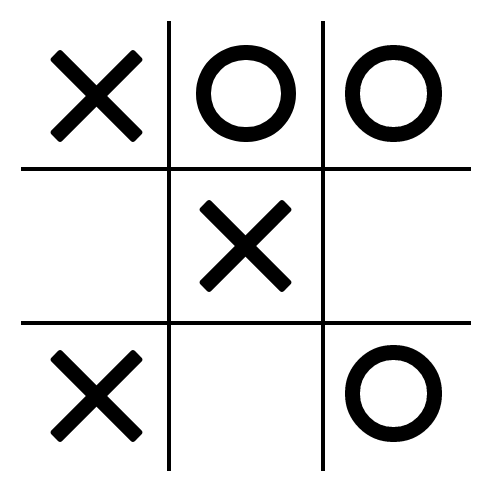
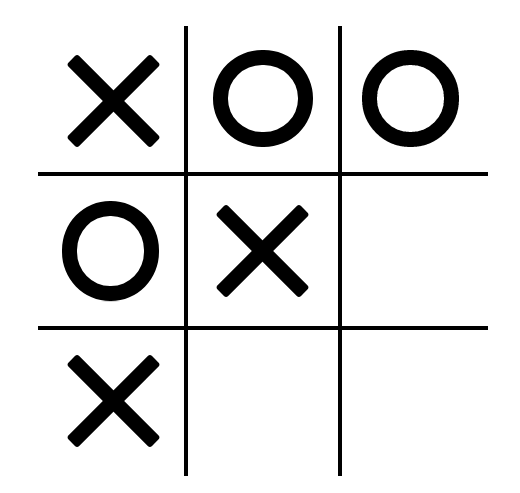
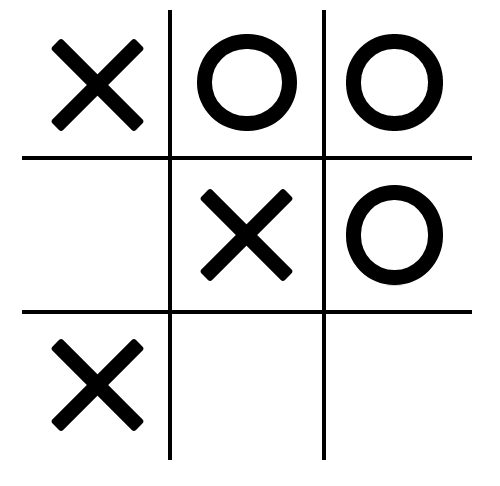
To see how this works for a specific position of a game, let us take an example.

Consider the following position:

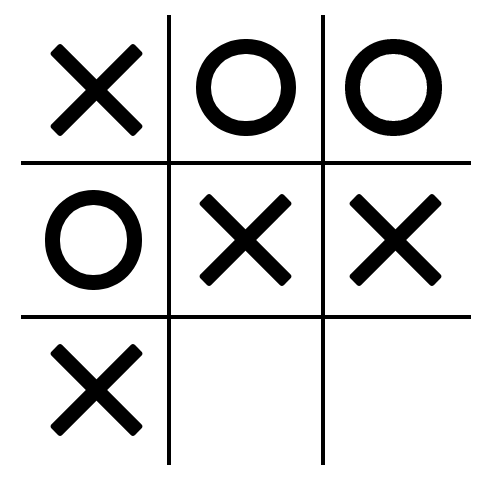
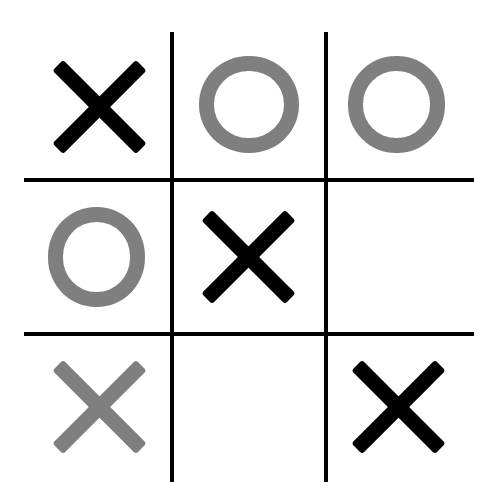
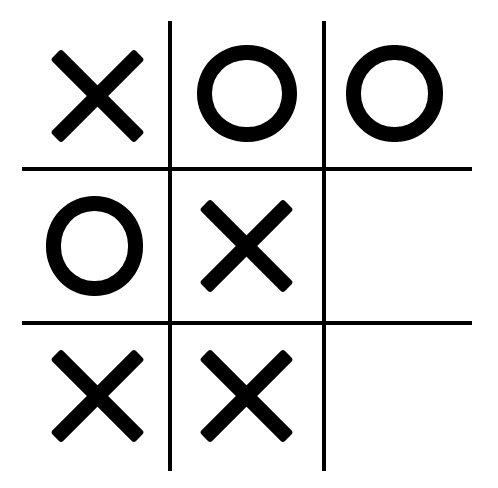


Now if you are familiar with the game, you will realise that the player ‘X’ (player 1) is going to win here, as there is no way for ‘O’ to block both the diagonal and the 1st column. Lets see how the computer comes to the same conclusion.

To calculate the values in the array ‘num’, the function recurse() will be called again with the player 2 having moved in all of his possible spots, so for the next iteration the following positions will be considered.

As none of these positions have a winner, the function recurse() will be called again, this time for the moves of player 1. Let us consider the next iteration for the first position.



As you can see, the second position will return an output of 1, while the others will return 0 (as they will result in a tie). So the values in array num will be {100,100,100,100,100,0,100,0,1},

or 100 | 100 |100

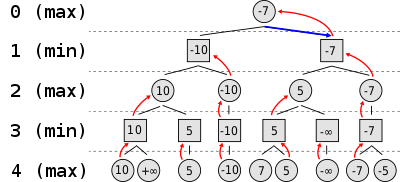
100 | 100 | 0

100 | 0 | 1

Here the turn is of the first player so it will return the maximum value (excluding 100), so it returns 1.

So the value stored in the num array of the previous recurse function will be {100,100,100,1,100,1,100,1,1}, the fourth element in this array is 1 because it was returned by the iteration explained above, and the rest of the ones are there because all of the playable options lead to the first player winning. Now as the turn is of player 2 here, it returns the minimum value, which is 1. So the function comes to the conclusion that player 1 is winning, as it returns 1 at the end. Note: to have a more comprehensive explanation, I have given the possible positions from the recurse function one depth at a time. In reality the program doesn’t actually work like that and searches through a whole ‘branch’ of possibilities before moving on to the next one.

Here is a representation of the minimax algorithm by a binary tree:



The minimax algorithm used in Tic-Tac-Toe works on a non-binary tree and the way I have implemented it, it only deals with the numbers 0, 1 or -1.

The function recurse() tells us which player is winning, but that doesn’t help us much to decide a move for the AI, to solve that we add a global variable ‘depth’ which tells us how many iterations deep the function recurse is. If depth becomes 0, we store the values of the array num in the global array ‘player1’ or ‘player2’, depending on whose turn it is. Then we can see which moves lead to what result (who will win) and choose a random move for the AI which leads to a victory, or a tie if no moves lead to the AI winning. The random element was added so that the AI doesn’t always play the same moves.

Variable Difficulty

To change the difficulty, a global variable ‘limit’ was introduced in the program. The user can set the value of the variable by changing the difficulty. ‘limit’ tells the program the maximum depth to which it can go in the recurse function. If the variable depth becomes equal to limit, the function returns 0 instead of calculating the return value further by calling itself again. So if the value of limit is set to 1, it will only call recurse function once more for every position and will assume that the move leads to a tie if no conclusion is reached.

Making the program more efficient:

4x4 Tic-Tac-Toe

Theoretically, we can use the same algorithm used in normal Tic-Tac-Toe to solve for 4x4 Tic-Tac-Toe. But when we actually try and run it, the computer hangs and no moves are made by the AI. This is because even though the size may only have changed by one, the complexity of the program changes exponentially and instead of 19863 possible combinations it becomes 3^16 (43046721) combinations. So to make a decent AI to play 4x4 Tic-Tac-Toe, we need to make the algorithm more efficient. This is done by two methods: alpha-beta pruning and variable depth.

Alpha-Beta Pruning

Alpha-beta pruning is a method used to increase the efficiency of the minimax algorithm. The basic concept of it is to remove any ‘nodes’ (possible board positions in the case of Tic-Tac-Toe) which the program has determined to be worse than any other move. In the 4x4 program alpha-beta pruning is implemented at a very primitive level. We know that in any situation, if we can find a move that connects 4-in-a-row for us, we should take it, and if we cannot do that, we should see if the opponent has any chance of making a 4-in-a-row, and block that if we find it. So the function recurse is changed so that at the start it searches for a possible move by which it can immediately win (three ‘X’s in a line for player 1) and if it finds such a move it returns 1 or -1 depending on who is winning and doesn’t explore other possible moves. After that it also looks for any three of the opponent’s symbols in a row and only considers moving there (to block the opponent from finishing the game) and no other possible moves. Furthermore, the program also stops evaluating other moves if calling recurse on a specific position returns 1 for player 1 or -1 for player 2, as that means it has found a winning move. These optimizations alone make the program 8-9 times faster, but that is not enough for it to completely solve 4x4 Tic-Tac-Toe completely. So we need to apply variable depth.

Variable Depth

In a game of Tic-Tac-Toe, the first few positions are always the hardest to solve, as you have to go through the most iterations. Also coincidentally, in 4x4 Tic-Tac-Toe the positions of the first few moves doesn’t matter as much, because you can always tie with the opponent regardless of what you moved in the first 2-3 moves. Keeping this in mind, we can reduce the value of the global var limit to 1 for the first few moves and then increase it according to how many moves have been played. Doing this causes no difference in the playing ability of the AI but makes it much faster. As ‘limit’ is used to make the program more efficient, difficulty is implemented by having a probability of playing a random move instead of one given by the AI in 4x4, 5x5 and 7x7 Tic-Tac-Toe.

5x5 Tic-Tac-Toe:

Predetermined Moves

5x5 Tic-Tac-Toe is a bit more tricky as you cannot let your first few moves randomly because that can give your opponent the opportunity to win. So to make an effective AI player for 5x5, the first 5 moves are predetermined by if else conditions. For example, if the opponent moves first and moves in the centre, the AI is programmed to choose a random adjacent diagonal square. If it had moved somewhere else, the opponent would have gotten a chance to play in such a way which would have ensured his victory. This is similar to the chess openings that chess masters remember instead of deciding on the spot, although much less complicated. After the first 5 moves are completed, the rest of the moves are decided by the same optimised minimax algorithm used in the 4x4 Tic-Tac-Toe.

7x7 Tic-Tac-Toe:

Restricting the search tree and float value return type

In 7x7 Tic-Tac-Toe, we cannot simply make the first few moves predetermined and solve the rest by minimax due to the much larger size. The minimax algorithm cannot be applied effectively unless the number of possible moves is pretty low (around 20), and we cannot solve many positions by predetermined moves because they will increase in complexity and will take a lot of time to code. So we need a way to restrict the number of positions the AI checks.

To implement this we need a way to differentiate ‘good’ moves from ‘bad’ moves. To do this I created a function ‘run’ which returns a list of all of the moves it considers ‘good’ in array form. To determine which move is good, the function plays every move one by one on a given board position and sends that new position to functions ‘attack’ and ‘block’. For every position given to the function ‘attack’, it returns an array of length 5 which counts the number of sequences of length 1, length 2, length 3, length 4 and length 5 of the player’s symbols (which aren’t blocked by the opponent) and stores them to the respective positions in the array to be returned. Similarly, the function 'blocks' returns the number of sequences of the opponent of respective length blocked by the player. After these functions are called, a ‘moveValue’ for every move is calculated by multiplying the individual attacks and blocks by specific weights and then adding them. If an option was available for an ‘attack’ of length 5 then it was taken as the only ‘good’ move. Otherwise the program searches for any move which can block 4-in-a-row of the opponent and returns that as the only ‘good’ move if found. If even that isn’t available the program then looks for any moves that make a double attack, which are searched by seeing if a move creates two different 4-length sequences at once. In a situation where none of these options are possible the program calls on a function ‘bestMoves’ which returns an array of all moves who’s moveValue is more than 0.5 times the maximum moveValue of any move. If that still is a lot of possible moves (more than 8) then the function returns the top 8 best moves. The weights chosen for each attack and block are powers of 4 and are proportional to the square of the number of elements in that position.

Choosing the maximum value move from all of the possible moves already creates a formidable opponent, but it still can be easily beaten if you plan your moves. So the set of moves considered 'good' by the function run() are then used in the minimax algorithm, which reduces the search tree by a lot and makes it much faster. It still takes the program a lot of time to solve it completely from the start, so variable depth is still required.

Float return type of minimax function

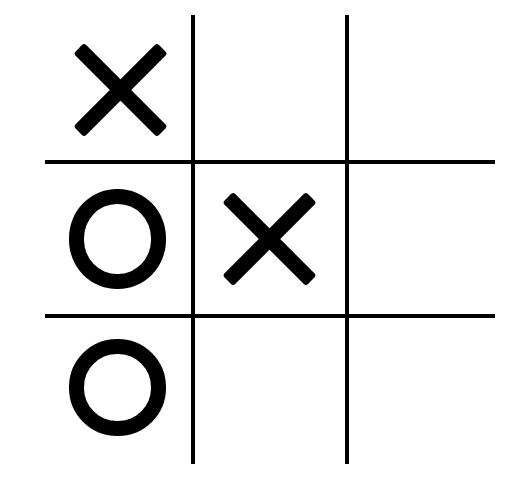
When the depth of the recurse function becomes equal to the value of limit, in the previous versions of the minimax algorithm we simply returned 0 for tie, but to make the program work better we can return a floating point value from -1 to 1 according to which player is at an advantage at that position. We can do this pretty easily as we already have the function 'attack' which we can use again. To calculate advantage we call a function 'decideAdvantage()', which calls the function 'attack()' for both players and multiplies them with specific weights and subtracts one from the other. In this case the weights are between 0 and 1 to ensure that the net sum is between -1 and 1.

Further Improvements and Possible Solutions

While the algorithms used in the program can be fun for casual playing and might win a few rounds against unskilled opponents, they still are very far from being perfect, both in the sense of efficiency as well as decision making.

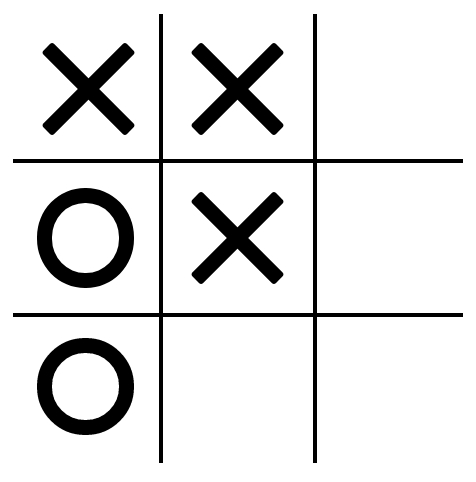
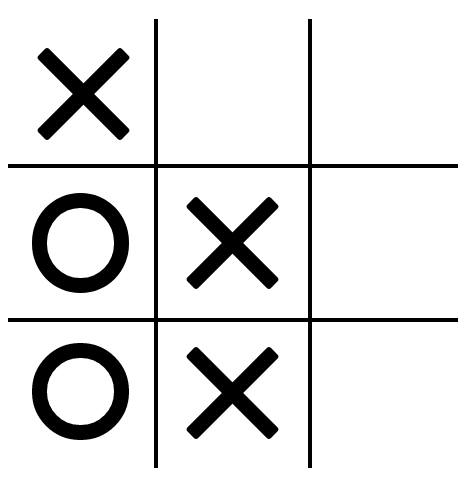
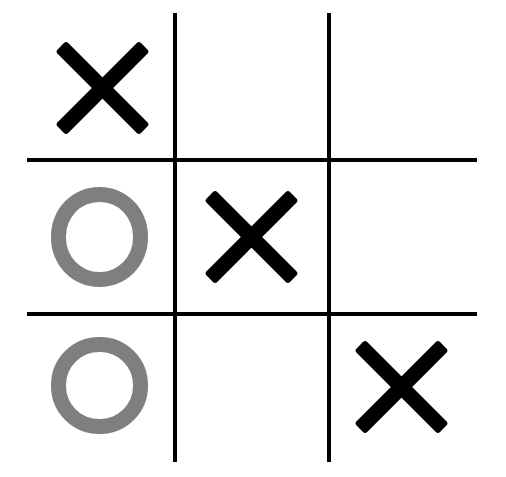
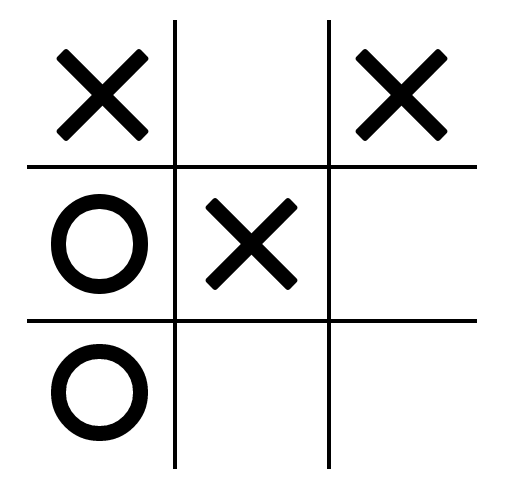
Problems with decision making

You might notice a few instances where the AI decides on moves which don’t make much sense. An example of this would be a situation like this:



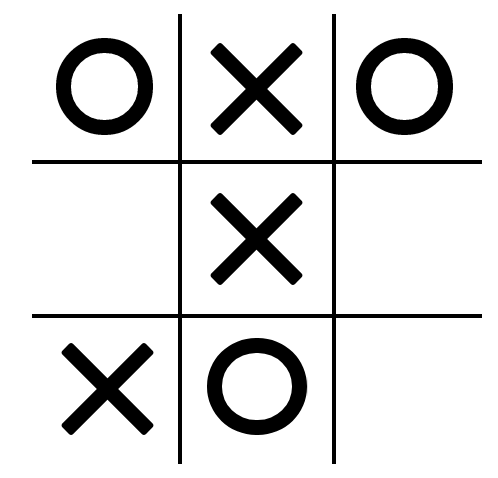
(player ‘X’ turn)

Now as a human, the choice here is obvious, simply put a cross on the bottom right corner and win, but to the computer all of these moves are equally valid:



This is because the computer still thinks that it is eventually going to win even if it plays these moves, and it doesn’t matter if it wins right now or after a hundred moves. This specific case may be solved by simple alpha-beta pruning which I have done for the 4x4 version, but a better and more general solution might be to also count the depth of the minimax algorithm at which a player wins. Then we can make the AI choose a move which makes it win at the lowest depth or the fewest moves.

Another problem that comes up is when the AI reduces its chances of victory because of its assumption that the opponent is a perfect player. Consider the following position:



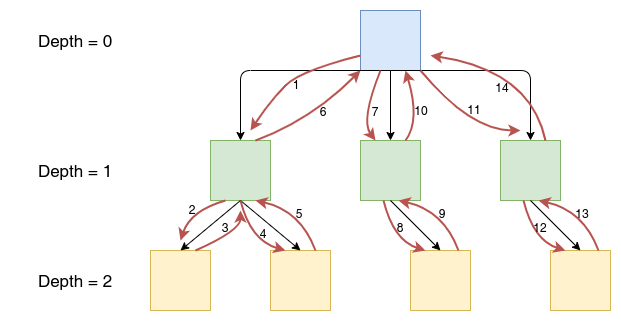
As a human, I would play in the middle row, so that if the opponent makes a mistake I can win, but for the computer all moves are equally good as it assumes that the opponent will block his attempt. To solve this problem we can look at the next iteration of the minimax algorithm and see if any possible moves that ‘O’ makes lead to ‘X’’s victory. Instead of just relying on the return value of the next function call of recurse(), we can also make the function return the ‘best case scenario’ as well as the ‘worst case scenario’, the program will assume the worst case scenario but if both worst case scenarios are the same it will also consider the best case scenarios.

Problems with efficiency

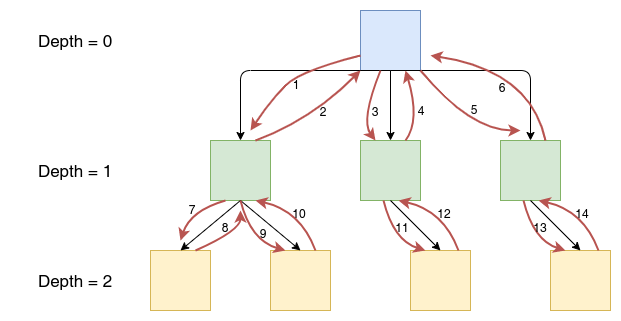
Iterative Deepening

As you might have noticed during playing the game, in some cases the AI plays a move immediately and in other cases (especially with the larger board sizes) the AI can take upto a few seconds to decide. This problem was reduced a bit by adding variable depth but you cannot determine what depth the AI must go to (to find the ideal balance of time and good decision making) just by looking at how many moves have been played. A simple solution might come to mind for this problem, how about we just add a timer, say of 0.5 seconds, and terminate the program when the timer ends and then look at what all decisions it has come up with in that time. But unfortunately this doesn’t work with the current version of minimax algorithm. Suppose a situation where the first few moves that the AI searches through are pretty bad and the very last move is the only one that assures a victory. In this situation if we have added a timer, what might happen is that the AI will completely search the branches of the first few moves and never make it to the last move before the timer ends and therefore not make a good decision. This is where iterative deepening comes to play. Instead of searching through a whole branch first, we can look at every move in the next iteration, and then every move in the iteration after that for better efficiency.

So instead of traversing the tree like:



It traverses like:



Storing Positions in Memory

One other way the algorithm can be made faster is by storing the result calculated for each different position in memory. What happens while iterating through all of the possible plays in the game is that sometimes the same position ends up getting solved for twice. For example if the first player plays at the centre position, second player plays at the top left corner, and then first player plays at the bottom right corner, it will result in the same position as the first player playing at the bottom right corner, second player plays at top left corner, and then first player plays at the centre spot. The computer will have to solve for this position twice. A possible solution would be saving these positions in an array and if a specific position is ever solved then the memory for that array will be changed and the results from solving that position will be added. To make this as efficient as possible we need a way to quickly locate a specific position in an array instead of searching each element of the array. To do this we can treat the board as a number in base 3 of length 9. The empty spots can be 0, player 1 spots can be 1 and player 2 spots can be 2. This will result in every possible position to get an unique number which can be its location in an array. The length of the array will be 3^9 = 19683, out of these there will be many positions which are not actually possible, for example 3 ‘X’s and no ‘O’s, but as the memory occupied isn’t that much it doesn’t matter (the actual number of possible positions are (½)Σ9Cn\*(n!)/([n/2]!\*[(n+1)/2]!) = 3023, where [ ] is floor function).

Another possible optimisation would be to relate similar positions and use the same data for them. For example one ‘X’ at the top right corner is the same as one ‘X’ at any other corner. These positions are exactly the same position, but rotated or flipped, so we can use the same data for them. So what we can do is that for every position solved, we fill the result in not only the array position for that result, but also the array position of all of its rotations and reflections (every position will have 7 other combinations if no axis of symmetry is present). This reduces the number of overall combinations to only 280. A benefit of saving everything in memory is that you also have to compute everything only once at the start and the rest of the game can be played from memory only.

These changes make the 3x3 Tic-Tac-Toe upto 65 times faster (only storing positions in memory, not iterative deepening) than the normal Tic-Tac-Toe minimax algorithm. The overall advantage in efficiency will be even greater as you will only have to run the algorithm once and can play as many games you want after that. Although I haven’t added these changes to the final code because the 3x3 version already solves any position in milliseconds and if we implement these changes to 4x4 or 5x5 then they will take up a lot of memory. A possible implementation will be by using hash tables and only storing positions that are expected to be repeated many times (maybe by using it along with iterative deepening we can store all of the positions in one specific depth in the hash table and deleting them later when we go to the next depth, hence not using too much memory).