TIC TAC TOE

Submitted by

Mayank Kumar Sohanda [RA2011051010048]

Vedant Singh [RA2011051010061]

Mihir Gupta [RA2011051010062]

Under the Guidance of

Dr. Ramprasath M
Assistant Professor,
Department of DSBS

In partial satisfaction of the requirements for the degree of

BACHELOR OF TECHNOLOGY
in
COMPUTER SCIENCE ENGINEERING

with specialization in GAMING TECHNOLOGY

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

SCHOOL OF COMPUTING

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR - 603203

JUNE 202

SRM INSTITUTION OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR-603203

BONAFIDE CERTIFICATE

Register No. RA2011051010048

Certified to be the bonafide record of project work done by **Mayank Kumar Sohanda (RA2011051010048)** of Computer Science Engineering with specialization in Gaming Technology, X1 section, IVth Semester, IInd Year B. Tech Degree course in the Practical 18CSC204J-Design and Analysis of Algorithm in SRM Institute of Science and Technology, Kattankulathur during the academic year 2021-2022 on the topic of "**TIC TAC TOE GAME**".

SUPERVISOR
Dr. Ramprasath M.
Assistant Professor
Department of DSBS

H.O.D
Dr. M. Lakshmi
Department of DSBS

Submitted for University Examination held in__________________ SRM Institute of Science and Technology, Kattankulathur.

Examiner 1

Examiner 2

Acknowledgement

We sincerely thank Mr. Ramprasath M Sir, who has been the greatest inspiration and has provided sufficient background knowledge and understanding of this subject. Our humble prostration goes to him, for providing all the necessary resources and environment, which has aided us to complete this project successfully.

Abstract

The MiniMax algorithm is a recursive algorithm used in decision-making and game theory. It delivers an optimal move for the player, considering that the competitor is also playing optimally. This algorithm is widely used for game playing in Artificial Intelligence, such as chess, tic-tac-toe, and myriad double players games.

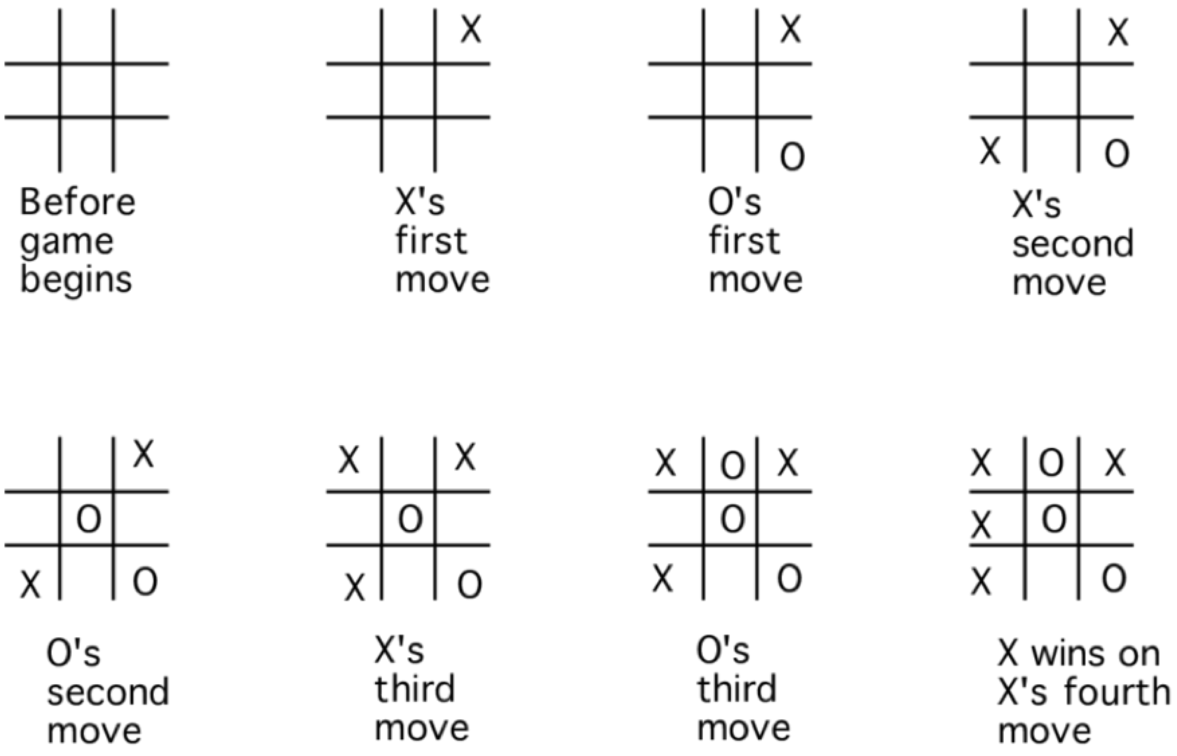
In this algorithm, two players play the game; one is called ***‘MAX’***, and the other is ***‘MIN.***’ The goal of players is to minimize the opponent’s benefit and maximize self-benefit. The MiniMax algorithm conducts a depth-first search to explore the complete game tree and then proceeds down to the leaf node of the tree, then backtracks the tree using recursive calls.

Problem Statement

To write a proper Tic-Tac-Toe AI (Artificial Intelligence) that plays a perfect game. This AI will consider all possible scenarios and makes the most optimal move.

The Game

Tic-tac-toe is a very popular game for two players, X and O, who take turns marking the spaces in a 3×3 grid. The player who succeeds in placing three of their marks in a vertical, horizontal or diagonal row wins the game.



Mathematical Properties

From a mathematical point of view the game has two very important properties:

Property 1:
The game admits the player that uses this **optimal strategy** will win or draw but it will not lose.

Property 2:
The number of possible different matches is relatively small.

At the start, the first player can mark any of the 9 spaces. In the following turn the second player can mark one of the remaining 8 spaces and so on. The game continues until all the spaces are marked or one of the players win.

It is then easy to understand that the total number of different matches is lower than:

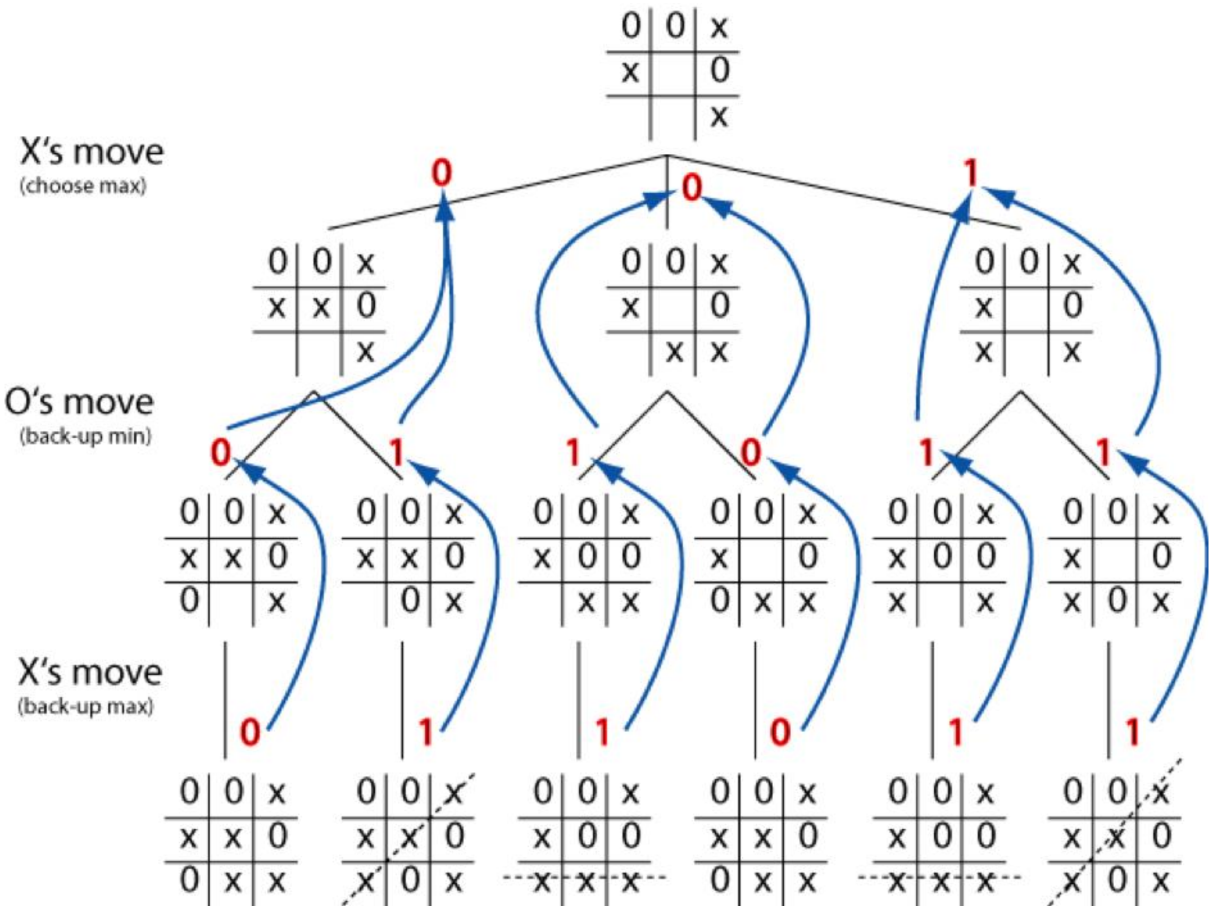
9*8*7....1 = 9! = 362880

That is a reasonably small number for a computer.

Algorithm

From properties 1 and 2 it follows that a practical, and general, algorithm to win/draw the game is to use the **Alpha Beta search**.

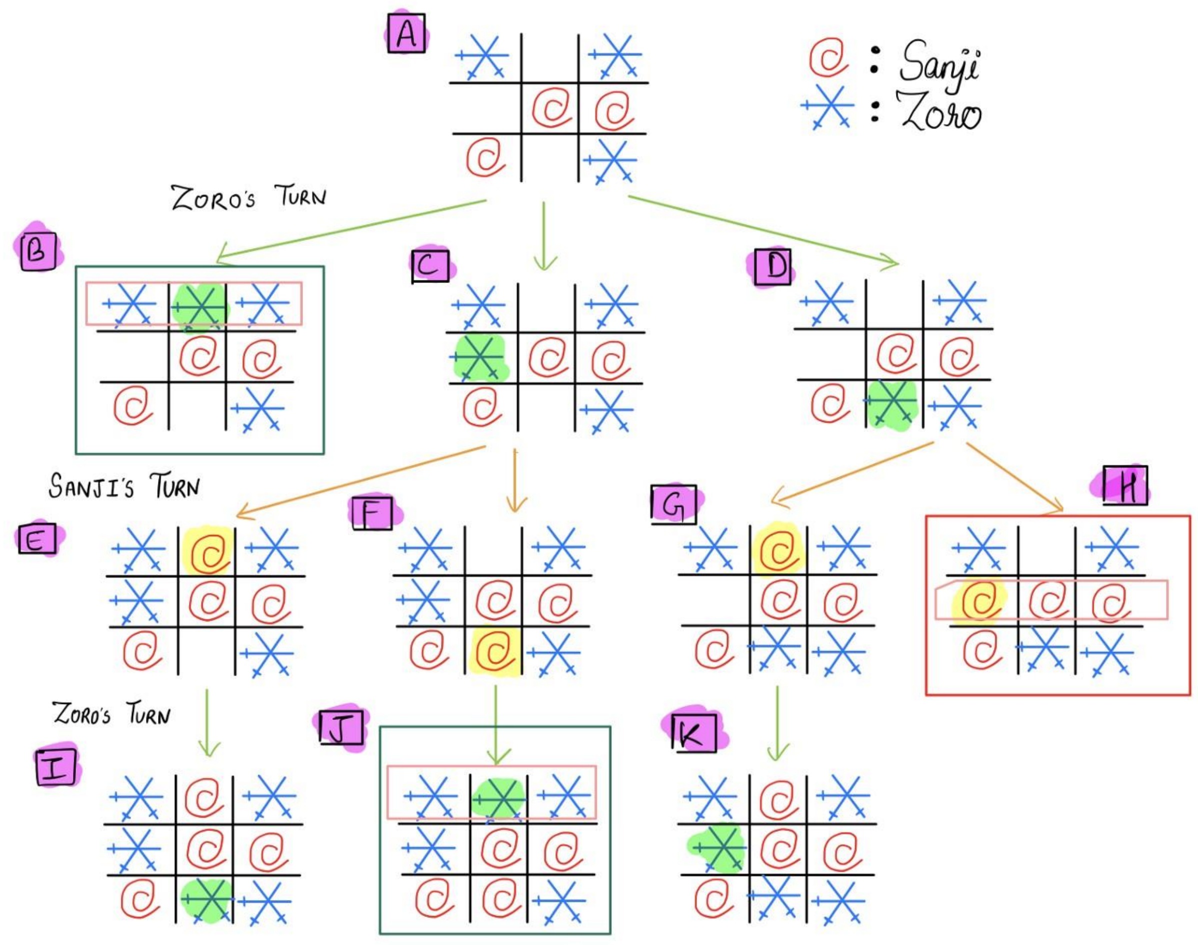
At each turn the algorithm evaluates all the possible consequences of each move (possible due to property 2) and chooses the one that will ensure a victory or a draw (possible due to property 1).



An AI player that chooses each move with the alpha beta search algorithm will never lose. To make the game more realistic it is nice to introduce a stochastic factor so that each time with a predefined probability the AI player moves randomly rather than following the alpha beta algorithm. This will make the game more realistic as it will make the AI player more human and sometimes it will lose.

Illustration

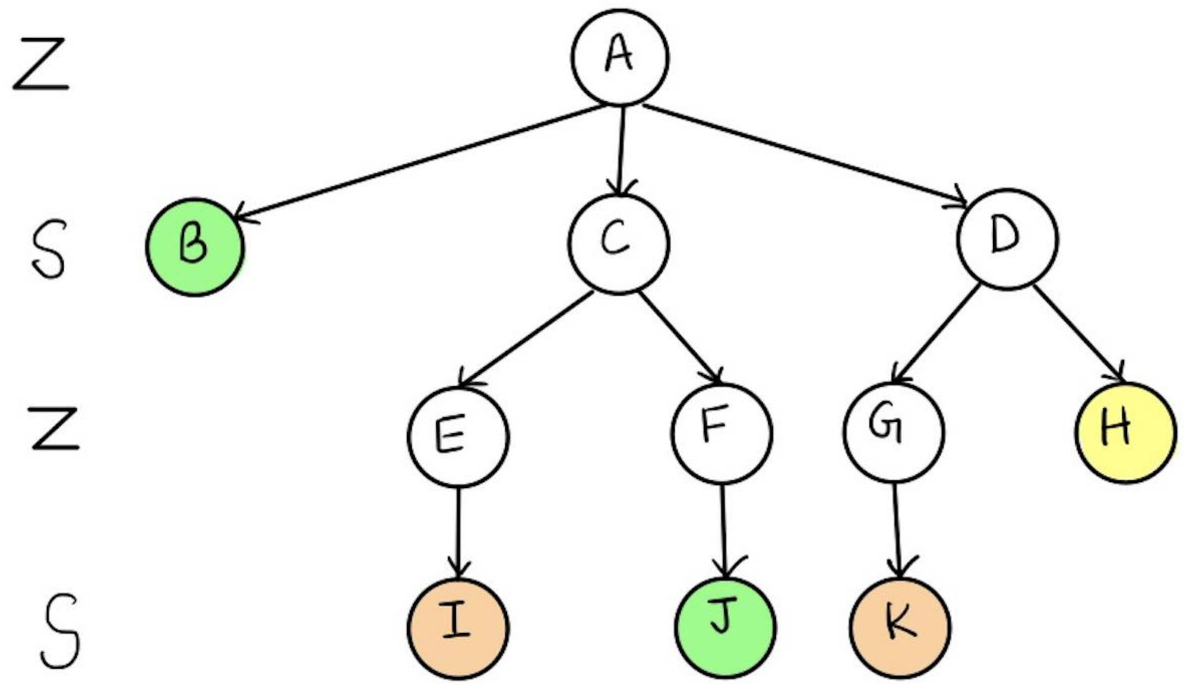
To begin with, we are considering two rivals, Zoro and Sanji, from the One Piece (a famous Japanese Manga series). They decided to play the tic-tac-toe to resolve an argument. Zoro starts the game by picking 'X' and Sanji picks 'O' and plays second. In this example, Zoro will try to get the maximum possible score, and Sanji will try to minimize the possible score of Zoro. Figure 1 shows the game tree after an intermediate stage.



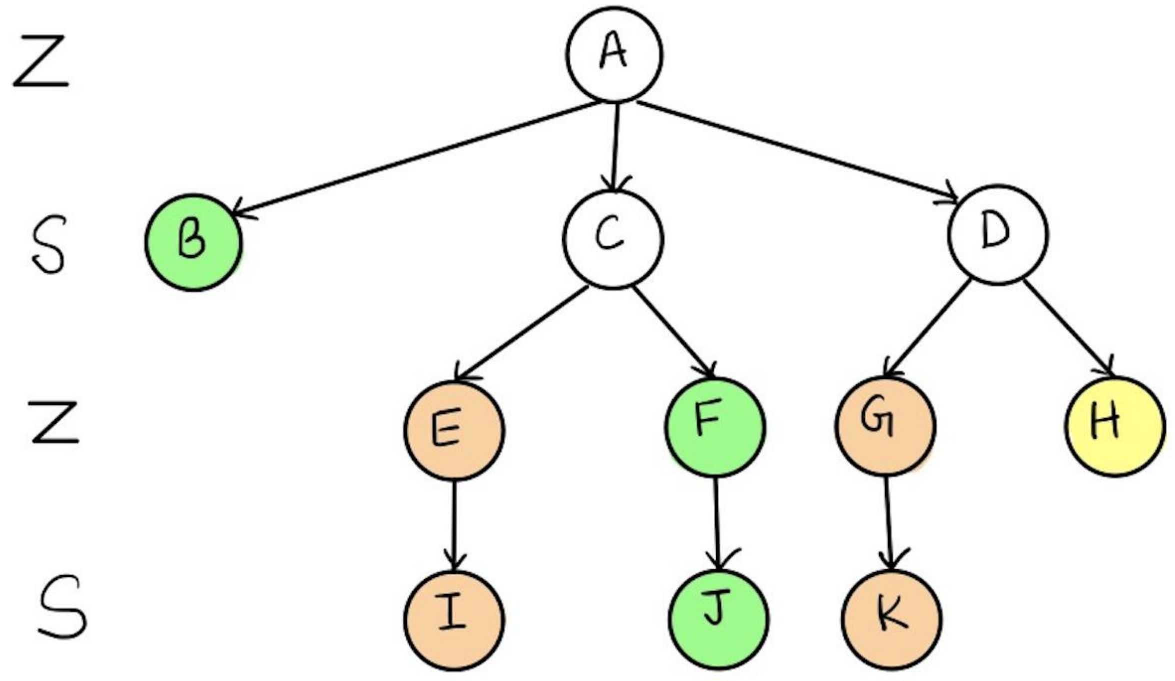
During an intermediate state (A) in the game, Zoro has three choices to pick from, i.e., state *'B,' 'C,'* and *'D.'* Picking state *'B'* will lead to the victory of Zoro, while choosing *'C'* and *'D'* will result in further rounds of the game. In case Zoro attempts for the *'C'* and *'D'* state, Sanji will play accordingly, and so on. There will be scenarios when both win disjointly as Sanji wins in state *'H,'* and Zoro wins in state *'J.'* The other leaf nodes will result in a draw.

At the leaf node, we give values for the backtracking purpose until the initial state is reached. For instance, if Zoro wins, we give +1 (*GREEN*); if Sanji wins, then Zoro will get -1 (*YELLOW*), and 0 (*LIGHT BROWN*) in case of no winner.

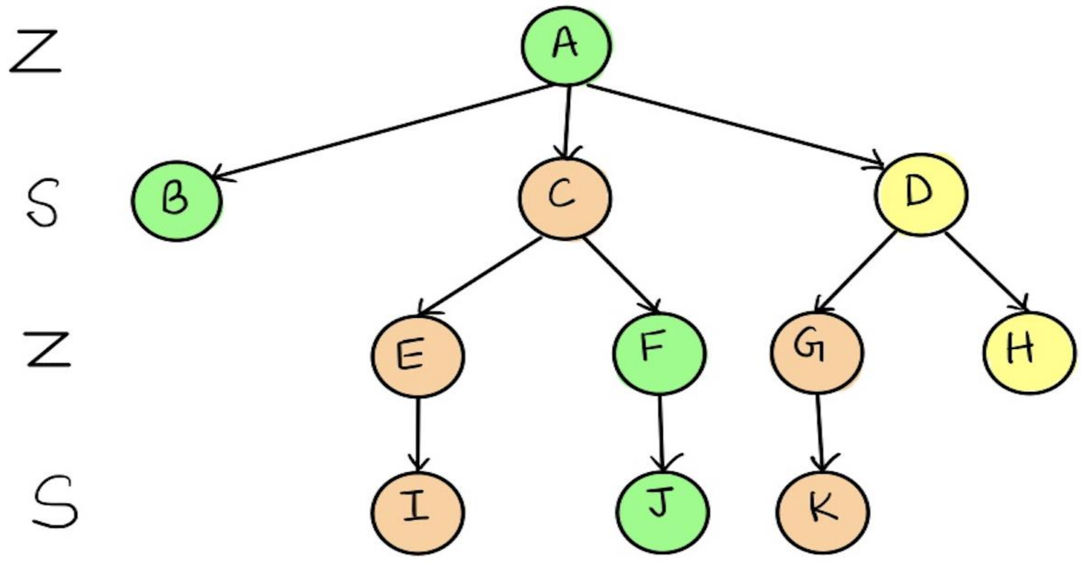
Following are the main steps involved in solving the two-player game tree:



Step 1: In the first step, the algorithm generates the entire game tree and applies the utility values (+1/-1/0) for leaf nodes. First of all, we will fill the utility value for node *'E.'* As it is Zoro's turn, he will try to maximize the score by picking 0 from *'I.'* Similarly, for node *'G,'* the utility value will be 0. For node *'F,'* Zoro will select +1. Figure 3 shows updated game tree.

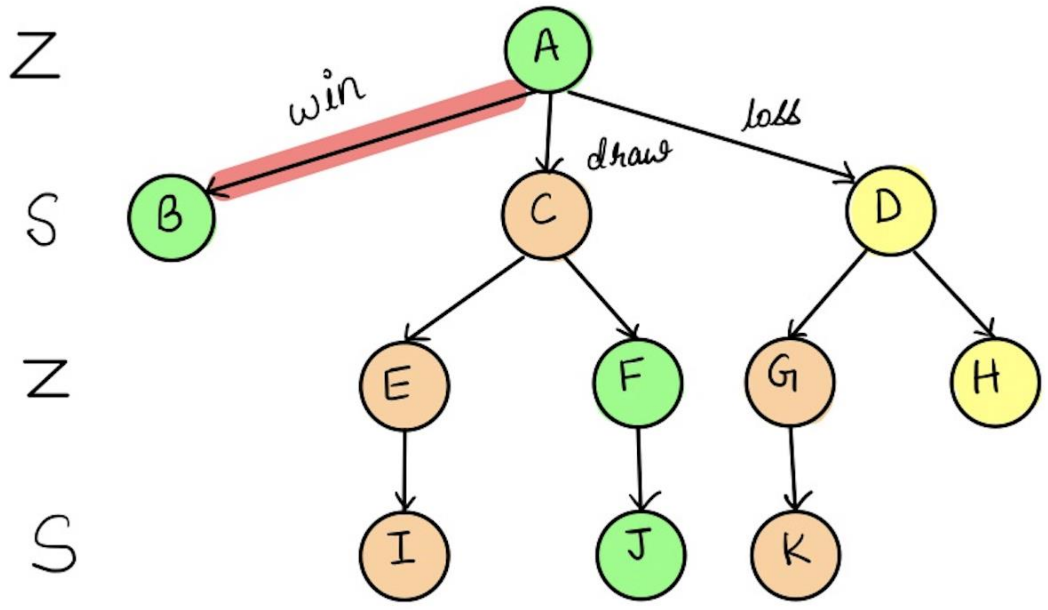


Step 2: Now, we will fill the utility values for 'C' and 'D.' As it is Sanji's turn so he will try to minimize Zoro's score, so for 'C,' he will pick a minimum of 'E' and 'F,' which is 0, and for 'D' he will pick minimum from 'G' and 'H,' which is -1. The updated game tree is shown in Figure 4.



Step 3: Figure 4 demonstrates the final tree, with each node having its utility values. At the level of node *'A,'* we have Zoro's turn, so he will pick the maximum from *'B,'* *'C,'* and *'D,'* which is +1. Hence, Zoro can win the game.

Therefore, in order to win after the mentioned given state, Zoro has one path that he can opt, i.e., *A → B,* which is highlighted as red in Figure 5. Path A → C will result in a draw, and Path A → D will result in Sanji's win.



Finding the best move

Finding the Best Move :

We shall be introducing a new function called **findBestMove()**. This function evaluates all the available moves using **minimax()** and then returns the best move the maximizer can make. The pseudocode is as follows :

```
function findBestMove(board):
    bestMove = NULL
    for each move in board :
        if current move is better than bestMove
            bestMove = current move
    return bestMove
```

MiniMax

Minimax :

To check whether or not the current move is better than the best move we take the help of **minimax()** function which will consider all the possible ways the game can go and returns the best value for that move, assuming the opponent also plays optimally

The code for the maximizer and minimizer in the **minimax()** function is similar to **findBestMove()**, the only difference is, instead of returning a move, it will return a value. Here is the pseudocode :

```
function minimax(board, depth, isMaximizingPlayer):


    if current board state is a terminal state :
        return value of the board


    if isMaximizingPlayer :
        bestVal = -INFINITY
        for each move in board :
            value = minimax(board, depth+1, false)
            bestVal = max( bestVal, value)
        return bestVal


    else :
        bestVal = +INFINITY
        for each move in board :
            value = minimax(board, depth+1, true)
            bestVal = min( bestVal, value)
        return bestVal
```

Checking for Game Over

Checking for GameOver state :

To check whether the game is over and to make sure there are no moves left we use **isMovesLeft()** function. It is a simple straightforward function which checks whether a move is available or not and returns true or false respectively. Pseudocode is as follows :

```
function isMovesLeft(board):
    for each cell in board:
        if current cell is empty:
            return true
    return false
```

Making AI Smarter

Making our AI smarter :

One final step is to make our AI a little bit smarter. Even though the following AI plays perfectly, it might choose to make a move which will result in a slower victory or a faster loss. Lets take an example and explain it.
Assume that there are 2 possible ways for X to win the game from a give board state.

- Move **A** : X can win in 2 move
- Move **B** : X can win in 4 moves

Our evaluation function will return a value of +10 for both moves **A** and **B**. Even though the move **A** is better because it ensures a faster victory, our AI may choose **B** sometimes. To overcome this problem we subtract the depth value from the evaluated score. This means that in case of a victory it will choose a the victory which takes least number of moves and in case of a loss it will try to prolong the game and play as many moves as possible. So the new evaluated value will be

- Move **A** will have a value of +10 – 2 = 8
- Move **B** will have a value of +10 – 4 = 6

Now since move **A** has a higher score compared to move **B** our AI will choose move **A** over move **B**. The same thing must be applied to the minimizer. Instead of subtracting the depth we add the depth value as the minimizer always tries to get, as negative a value as possible. We can subtract the depth either inside the evaluation function or outside it. Anywhere is fine. I have chosen to do it outside the function. Pseudocode implementation is as follows.

```
if maximizer has won:
   return WIN_SCORE – depth


else if minimizer has won:
   return LOOSE_SCORE + depth
```

Code

```
// C++ program to find the next optimal move for
// a player
#include<bits/stdc++.h>
using namespace std;

struct Move
{
	int row, col;
};

char player = 'x', opponent = 'o';

// This function returns true if there are moves
// remaining on the board. It returns false if
// there are no moves left to play.
bool isMovesLeft(char board[3][3])
{
	for (int i = 0; i<3; i++)
		for (int j = 0; j<3; j++)
			if (board[i][j]=='_')
				return true;
	return false;
}

// This is the evaluation function as discussed
int evaluate(char b[3][3])
{
	// Checking for Rows for X or O victory.
```

```
for (int row = 0; row<3; row++)

{

        if (b[row][0]==b[row][1] &&

                b[row][1]==b[row][2])

        {

                if (b[row][0]==player)

                        return +10;

                else if (b[row][0]==opponent)

                        return -10;

        }

}


// Checking for Columns for X or O victory.

for (int col = 0; col<3; col++)

{

        if (b[0][col]==b[1][col] &&

                b[1][col]==b[2][col])

        {

                if (b[0][col]==player)

                        return +10;


                else if (b[0][col]==opponent)

                        return -10;

        }

}


// Checking for Diagonals for X or O victory.

if (b[0][0]==b[1][1] && b[1][1]==b[2][2])

{

        if (b[0][0]==player)

                return +10;
        else if (b[0][0]==opponent)
            return -10;
    }

    if (b[0][2]==b[1][1] && b[1][1]==b[2][0])
    {
        if (b[0][2]==player)
            return +10;
        else if (b[0][2]==opponent)
            return -10;
    }

    // Else if none of them have won then return 0
    return 0;
}

// This is the minimax function. It considers all
// the possible ways the game can go and returns
// the value of the board
int minimax(char board[3][3], int depth, bool isMax)
{
    int score = evaluate(board);

    // If Maximizer has won the game return his/her
    // evaluated score
    if (score == 10)
        return score;

    // If Minimizer has won the game return his/her
    // evaluated score
    if (score == -10)
```

```
        return score;

// If there are no more moves and no winner then
// it is a tie
if (isMovesLeft(board)==false)
        return 0;

// If this maximizer's move
if (isMax)
{
        int best = -1000;

        // Traverse all cells
        for (int i = 0; i<3; i++)
        {
                for (int j = 0; j<3; j++)
                {
                        // Check if cell is empty
                        if (board[i][j]=='_')
                        {
                                // Make the move
                                board[i][j] = player;

                                // Call minimax recursively and choose
                                // the maximum value
                                best = max( best,
                                        minimax(board, depth+1, !isMax) );

                                // Undo the move
                                board[i][j] = '_';
                        }
```

```
            }
        }
        return best;
    }

    // If this minimizer's move
    else
    {
        int best = 1000;

        // Traverse all cells
        for (int i = 0; i<3; i++)
        {
            for (int j = 0; j<3; j++)
            {
                // Check if cell is empty
                if (board[i][j]=='_')
                {
                    // Make the move
                    board[i][j] = opponent;

                    // Call minimax recursively and choose
                    // the minimum value
                    best = min(best,
                        minimax(board, depth+1, !isMax));

                    // Undo the move
                    board[i][j] = '_';
                }
            }
        }
```

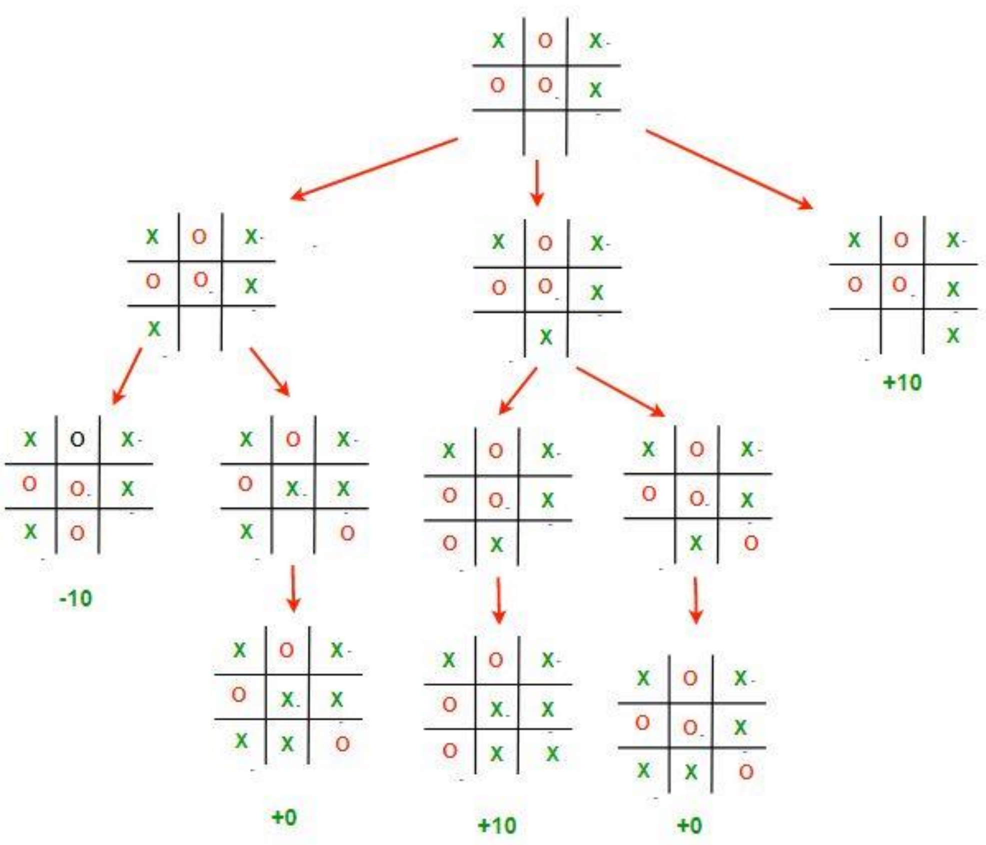
```
			return best;
		}
}

// This will return the best possible move for the player
Move findBestMove(char board[3][3])
{
		int bestVal = -1000;
		Move bestMove;
		bestMove.row = -1;
		bestMove.col = -1;

		// Traverse all cells, evaluate minimax function for
		// all empty cells. And return the cell with optimal
		// value.
		for (int i = 0; i<3; i++)
		{
			for (int j = 0; j<3; j++)
			{
				// Check if cell is empty
				if (board[i][j]=='_')
				{
					// Make the move
					board[i][j] = player;

					// compute evaluation function for this
					// move.
					int moveVal = minimax(board, 0, false);

					// Undo the move
					board[i][j] = '_';
```

```
                    // If the value of the current move is
                    // more than the best value, then update
                    // best/
                    if (moveVal > bestVal)
                    {
                        bestMove.row = i;
                        bestMove.col = j;
                        bestVal = moveVal;
                    }
                }
            }
        }

        printf("The value of the best Move is : %d\n\n",
                bestVal);

        return bestMove;
}

// Driver code
int main()
{
        char board[3][3] =
        {
            { 'x', 'o', 'x' },
            { 'o', 'o', 'x' },
            { '_', '_', '_' }
        };

        Move bestMove = findBestMove(board);
        printf("The Optimal Move is :\n");
        printf("ROW: %d COL: %d\n\n", bestMove.row,
                                                        bestMove.col );
        return 0;
}
```

Output :

```
The value of the best Move is : 10


The Optimal Move is :
ROW: 2 COL: 2
```

Explanation



This image depicts all the possible paths that the game can take from the root board state. It is often called the **Game Tree**.

The 3 possible scenarios in the above example are :

- **Left Move** : If X plays [2,0]. Then O will play [2,1] and win the game. The value of this move is -10
- **Middle Move** : If X plays [2,1]. Then O will play [2,2] which draws the game. The value of this move is 0
- **Right Move** : If X plays [2,2]. Then he will win the game. The value of this move is +10;

Remember, even though X has a possibility of winning if he plays the middle move, O will never let that happen and will choose to draw instead.
Therefore the best choice for X, is to play [2,2], which will guarantee a victory for him.
We do encourage our readers to try giving various inputs and understanding why the AI choose to play that move. Minimax may confuse programmers as it it thinks several moves in advance and is very hard to debug at times. Remember this implementation of minimax algorithm can be applied any 2 player board game with some minor changes to the board structure and how we iterate through the moves. Also sometimes it is impossible for minimax to compute every possible game state for complex games like Chess. Hence we only compute upto a certain depth and use the evaluation function to calculate the value of the board.

Time complexity : O(b^d) b is the branching factor and d is count of depth or ply of graph or tree.

Space Complexity : O(bd) where b is branching factor into d is maximum depth of tree similar to DFS.

References

- Wikipedia
- GeeksForGeeks
- StackOverFlow