**Abstract**

Decision tree learning is one of the most successful techniques for supervised classification learning. Each internal node of a decision tree is labelled with an input feature. The arcs coming from a node labelled with a feature are labelled with each of the possible values of the feature.

Backtracking is an algorithmic-technique that solves a given problem using a recursive approach. It finds all possible combination to solve an optimization problem.

Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally. It is widely used in two player turn-based games such as Tic-Tac-Toe, Backgammon, Mancala, Chess, etc.

In Minimax the two players are called maximizer and minimizer. The **maximizer** tries to get the highest score possible while the **minimizer** tries to do the opposite and get the lowest score possible.

Every board state has a value associated with it. This value is different for every game and depends on the specifics and constrains the game offers. In a given state if the maximizer has upper hand then, the score of the board will tend to be some positive value. If the minimizer has the upper hand in that board state then it will tend to be some negative value.

**Problem Definition**

The problem statement here is to find the most optimal move in a game of tic tac toe. The objective of the game are simple. We need to get three in a row. The first player, in this case the maximizer is known as X and the second, the minimizer, is O. Players alternate placing X’ s and O’s on the game board until either opponent has three in a row or all nine squares are filled. Three in a row can mean a single row, column or any of the two diagonals.

The expected input would be a 3x3 matrix marked with X’s and 0’s and ‘\_’ signifying an empty space.

The output would be the most optimal move that a maximizer X can make.

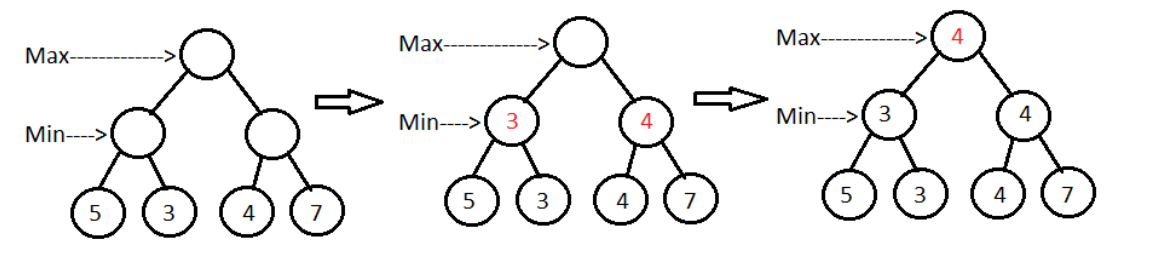
We first create a minimax decision tree, by using tools of recursion and computing all paths.

**Introduction**

The process of getting the optimal move in a game of tic tac toe is started by forming a minimax decision tree. Our goal is to find the best move for the player. To do so, we can just choose the node with best evaluation score. To make the process smarter, we can also look ahead and evaluate potential opponent’s moves.

For each move, we can look ahead as many moves as our computing power allows. The algorithm assumes that the opponent is playing optimally. The root node is chosen as the start and choose the best possible node based on their evaluation scores. The evaluation function can assign scores to only result nodes (leaves). Therefore, we recursively reach leaves with scores and back propagate the scores.

We first consider an example of a simple minmax tree:



Maximizer starts with the root node and chooses the move with the maximum score. As only leaves have evaluation scores with them, the algorithm reaches leaf nodes recursively. In the given game tree, currently it’s the minimizer’s turn to choose a move from the leaf nodes, so the nodes with minimum scores 3 and 4 will get selected. It keeps picking the best nodes similarly, till it reaches the root node. The same process applies for a game of tic tac toe except that there are rules and constraints the players must follow. Values at nodes are also calculated based on heuristics of the game.

**Design**

**Algorithm**

Here in the implementation of finding optimal path in a game of tic tac toe, we follow the following steps:

1. Construct the complete game tree
2. Evaluate scores for leaves using the evaluation function
3. Back-up scores from leaves to root, considering the player type:
4. For max player, select the child with the maximum score
5. For min player, select the child with the minimum score
6. At the root node, choose the node with max value and perform the corresponding move

**Minimax tree construction**

A game tree also known as minimax tree is constructed based on the following algorithm:

**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)

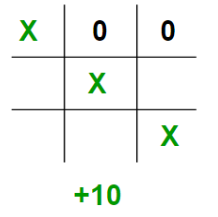
bestVal = min(bestVal, value)

**return** bestVal

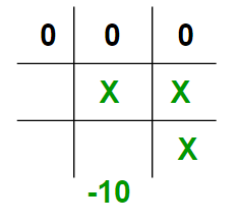
**Evaluation Function**

The task at hand is to generate a minimax tree for game of tic tac toe. The values are computed based on heuristics and values obtained by evaluation function.

If X wins on the board, we give it a positive value of +10.

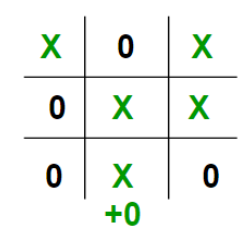


If O wins on the board, we give it a negative value of -10.



If no one has won or the game results in a draw then we give a value of +0

.



**Finding the best move**

The best move is found by recursively travelling all moves and obtaining the most optimal one. The pseudocode for the function is given below:

**function** findBestMove(board):

bestMove = NULL

**for each** move in board:

if current move is better than bestMove

bestMove = current move

**return** bestMove

**Implementation**

We implement the previously mentioned functions using C programming language.

‘Evaluate’ acts as the evaluation function for the minimax tree. We represent our board as a 3×3 2D character matrix, like char board [3][3]; then we have to check each row, each column and the diagonals to check if either of the players have gotten 3 in a row.

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]=='x')

               return +10;

            else if (b[row][0]=='o')

               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]=='x')

                return +10;

            else if (b[0][col]=='o')

                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]=='x')

            return +10;

        else if (b[0][0]=='o')

            return -10;

    }

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

    {

        if (b[0][2]=='x')

            return +10;

        else if (b[0][2]=='o')

            return -10;

    }

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

    return 0;

}

The above code considers all possible results of a given board and returns 10 if the maximizer wins, -10 if the minimizer wins and 0 if there’s a draw.

Based on our evaluation function we then construct a minimax tree.

int minimax(char board[3][3], int depth, int Max)

{

int score = evaluate(board);

if (score == 10)

return score;

if (score == -10)

return score;

if (MovesLeft(board)==0)

return 0;

// If this maximizer's move

if (Max)

{

int best = -1000;

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

{

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

{

if (board[i][j]=='\_')

{

board[i][j] = 'x';

best = max( best, minimax(board, depth+1, !Max) );

board[i][j] = '\_';

}

}

}

return best;

}

// minimizer's move

else

{

int best = 1000;

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

{

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

{

if (board[i][j]=='\_')

{

board[i][j] = 'o';

best = min(best, minimax(board, depth+1, !Max));

board[i][j] = '\_';

}

}

}

return best;

}

}

Given a board we use a function findBestMove to figure out the most optimal move:

int findBestMove(char board[3][3])

{

int bestVal=-1000;

struct Move bestMove;

bestMove.row=-1;

bestMove.col=-1;

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

{

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

{

if (board[i][j]=='\_')

{

board[i][j] = 'x';

int moveVal = minimax(board, 0,0);

board[i][j] = '\_';

if (moveVal > bestVal)

{

bestMove.row = i;

bestMove.col = j;

bestVal = moveVal;

}

}

}

}

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

printf("The best Move is: %d %d\n", bestMove.row,bestMove.col);

return 0;

}

This function prints out the move the maximizer takes optimally considering the opponent makes all choices optimally.

The time complexity for generation of minimax tree is O(b^m) and the space complexity is O(bm), where b is the number of legal moves at each point and m is the maximum depth of the tree. The time complexity for evaluate algorithm is O(n) where n is the number of columns/rows, in the case of tic tac toe n=3.

**Conclusion:**

In conclusion, we say that Minimax algorithm is one of the most popular algorithms for computer board games. It is widely applied in turn based games. It can be a good choice when players have complete information about the game.

It may not be the best choice for the games with exceptionally high branching factor (e.g. game of GO). Nonetheless, given a proper implementation, it can be a pretty smart AI.

Minmax are used in creating games such as GO, Chess, rock-paper-scissors, Kuhn poker, Dictator game , Rendezvous problem ,etc.

**Resources**

The following are the resources used for this project :

1. <https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/>
2. <https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-2-evaluation-function/>
3. <https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-3-tic-tac-toe-ai-finding-optimal-move/>