18CSC204J-Design and Analysis of Algorithms (DAA)
Project Report

TIC TAC TOE USING MINI-MAX ALGORITHM

Submitted by:

Under the Guidance of
Dr. Prabhu Kavin B
(Assistant Professor, Department of DSBS)

In partial fulfilment of the requirements for the degree of
BACHELOR OF TECHNOLOGY in
COMPUTER SCIENCE ENGINEERING
with specialization in Business Systems

DEPARTMENT OF DATA SCIENCE,
SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR - 603 203
NOVEMBER 2022

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR-603203

BONAFIDE CERTIFICATE

Certified to be the bonafide record of work done for 18CSC361J semester project report titled **"TIC TAC TOE USING MINI-MAX ALGORITHM"** by

who carried out the project under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

Date: **Lab Incharge:**

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

Examiner - 1   **Examiner - 2**

TABLE OF CONTENTS

1. Problem Definition:

Definition of tic-tac-toe:

It's a game in which two players alternately put Xs and Os in compartments of a figure formed by two vertical lines crossing two horizontal lines and each tries to get a row of three Xs or three Os before the opponent does.

History and Etymology for tic-tac-toe:

Tic-tac-toe, was a former game in which players with eyes shut brought a pencil down on a slate marked with numbers and scored the number hit.

In Artificial Intelligence and Game Theory, we encounter search problems that can be interpreted by a graph of interconnected nodes, each representing a possible state. An intelligent agent must traverse graphs by evaluating each node to reach an optimal state. Nevertheless, there are specific problems where standard graph search algorithms cannot be applied.

Describing a Perfect Game of Tic-Tac-Toe:

If I play perfectly, every time I play I will either win the game, or I will draw the game. Furthermore, if I play against another perfect player, I will always draw the game.

How might we describe these situations quantitatively? Let's assign a score to the "end game conditions:"

- I win, hurray! I get 10 points!

- I lose, shit. I lose 10 points (because the other player gets 10 points)

- I draw, whatever. I get zero points, nobody gets any points.

So now we have a situation where we can determine a possible score for any game-end state.

## 2. Problem Explanation with diagram and example:

To apply this, let's take an example from near the end of a game, where it is my turn. I am X. My goal here, obviously, is to maximize my end-game score.

Win! +10
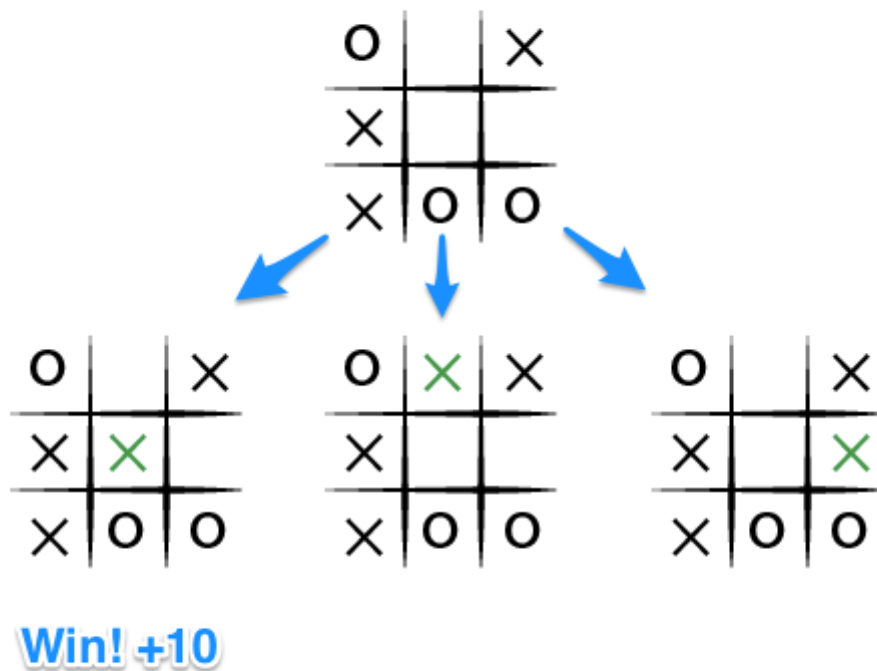
If the top of this image represents the state of the game I see when it is my turn, then I have some choices to make, there are three places I can play, one of which clearly results in me winning and earning the 10 points. If I don't make that move, O could very easily win. And I don't want O to win, so my goal here, as the first player, should be to pick the maximum scoring move.

But What About O?

What do we know about O? Well we should assume that O is also playing to win this game, but relative to us, the first player, O wants obviously wants to choose the move that results in the worst score for us, it wants to pick a move that would minimize our ultimate score. Let's look at things from O's perspective, starting with the two other game states from above in which we don't immediately win:
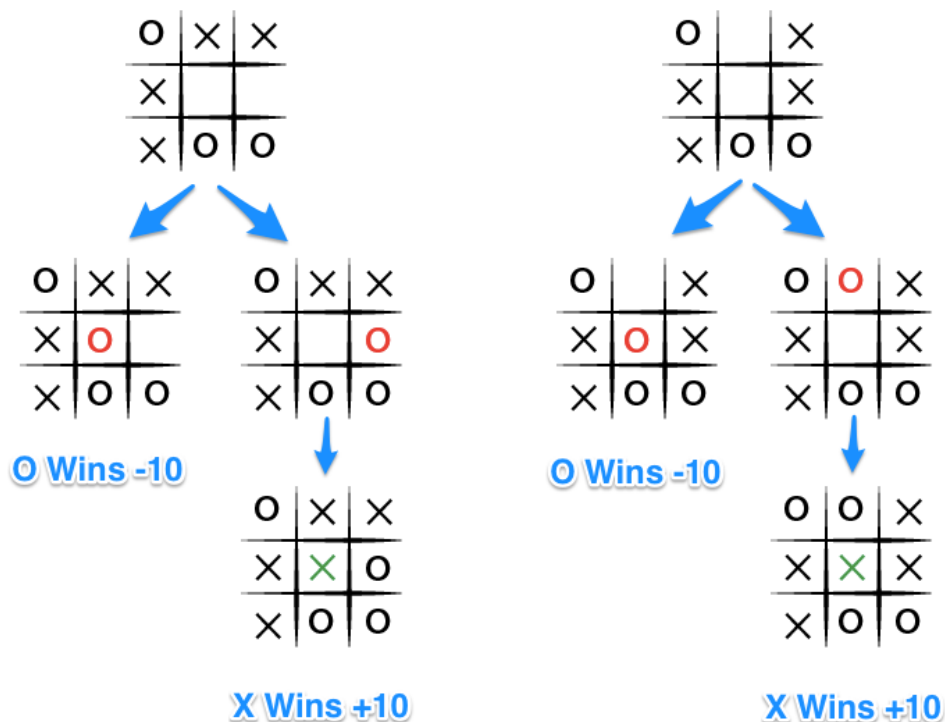
The choice is clear, O would pick any of the moves that result in a score of -10.

Describing Minimax:

The key to the Minimax algorithm is a back and forth between the two players, where the player whose "turn it is" desires to pick the moves with the maximum score. In turn, the scores for each of the available moves are determined by the opposing player deciding which of its available moves has the minimum score. And the scores for the opposing player's moves are again determined by the turn-taking player trying to maximize its score and so on all the way down the move tree to an end state.

A description for the algorithm, assuming X is the "turn-taking player," would look something like:

- If the game is over, return the score from X's perspective.
- Otherwise get a list of new game states for every possible move
- Create a scores list
- For each of these states add the minimax result of that state to the scores list
- If it's X's turn, return the maximum score from the scores list
- If it's O's turn, return the minimum score from the scores list

You'll notice that this algorithm is recursive, it flips back and forth between the players until a final score is found.

3. Design Techniques used:

Backtracking is a general algorithmic technique that considers searching every possible combination in order to solve an optimization problem. The basic principle of a backtracking algorithm, in regards to Sudoku, is to work forwards, one square at a time to produce a working Sudoku grid. When a problem occurs, the algorithm takes itself back one step and tries a different path. It's nearly impossible to produce a valid Sudoku by randomly plotting numbers and trying to make them fit. Likewise, backtracking with a random placement method is equally ineffective. Backtracking best works in a linear method. It is fast, and effective if done correctly.
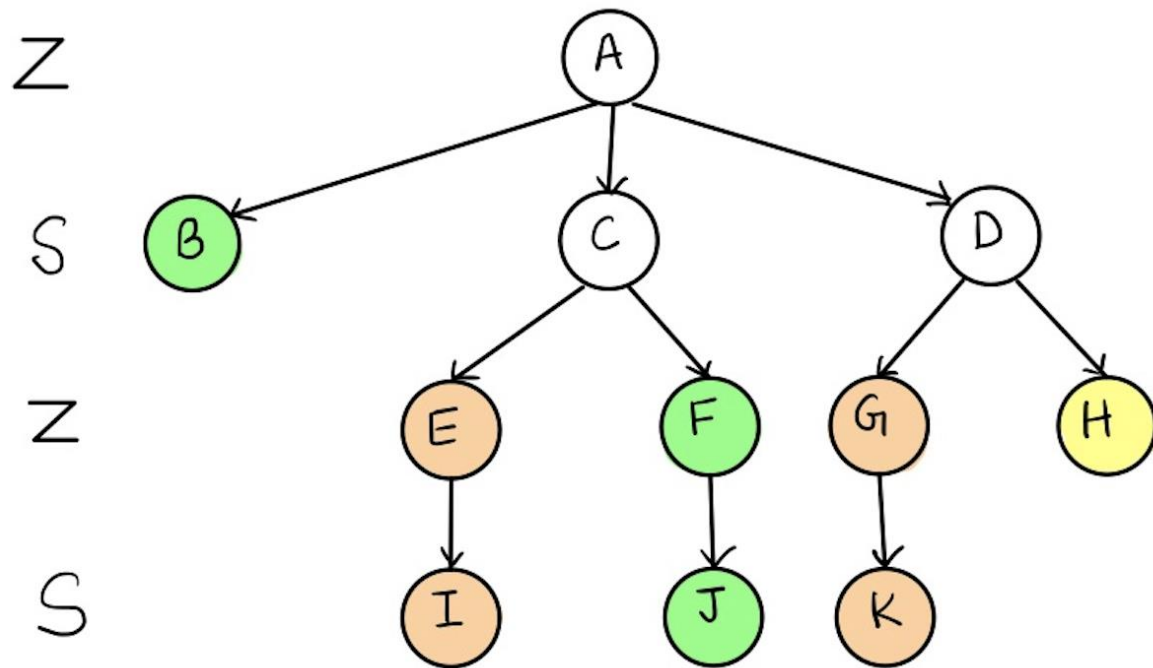
State Space representation of Tic-Tac-Toe

Each state is denoted by the positions occupied by the letters of the two players in a 3x3 matrix and other empty places.

1. **Initial State:** An empty 3x3 matrix.

2. **Intermediate State:** Any arrangement of a 3x3 matrix obtained after applying valid rules to the current state.

3. **Final State:** Same letters in a complete row or complete column, or complete diagonal win the game.

4. **Rules:** The players will get turns one after the other; they can mark their letters on the empty cell.

MiniMax Illustration using the Tic-Tac-Toe game

To begin with, we are considering two rivals, Zoro and Sanji, from 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.

4. Algorithm for the problem:

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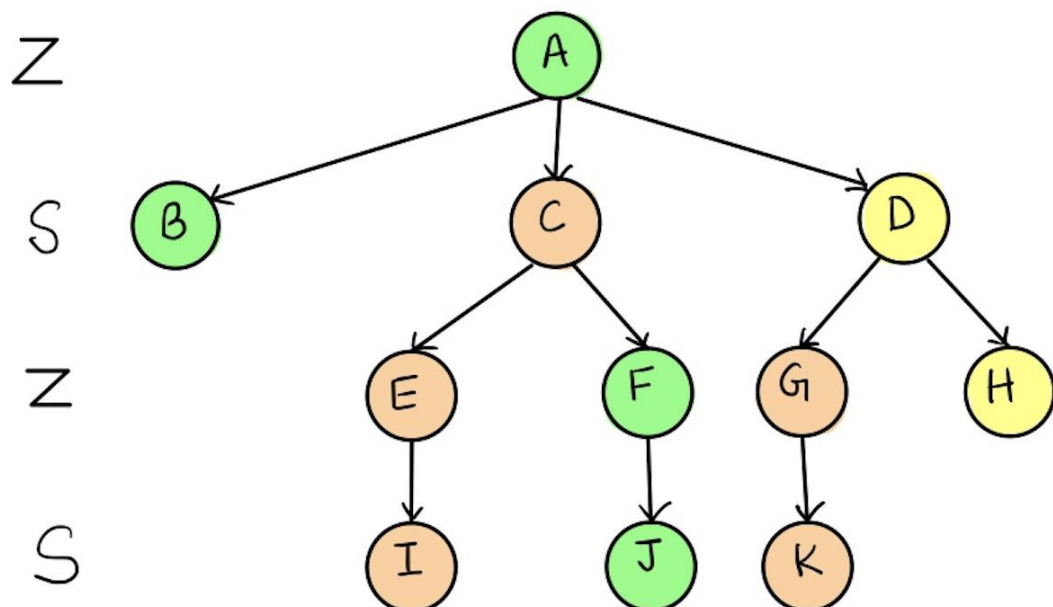
Step 3: Figure 4 demonstrates the final tree, with each node having its utility values. At the level of node 'A,' we have Zoro's turn, so he will pick the maximum from 'B,' 'C,' and 'D,' which is +1. Hence, Zoro can win the game.

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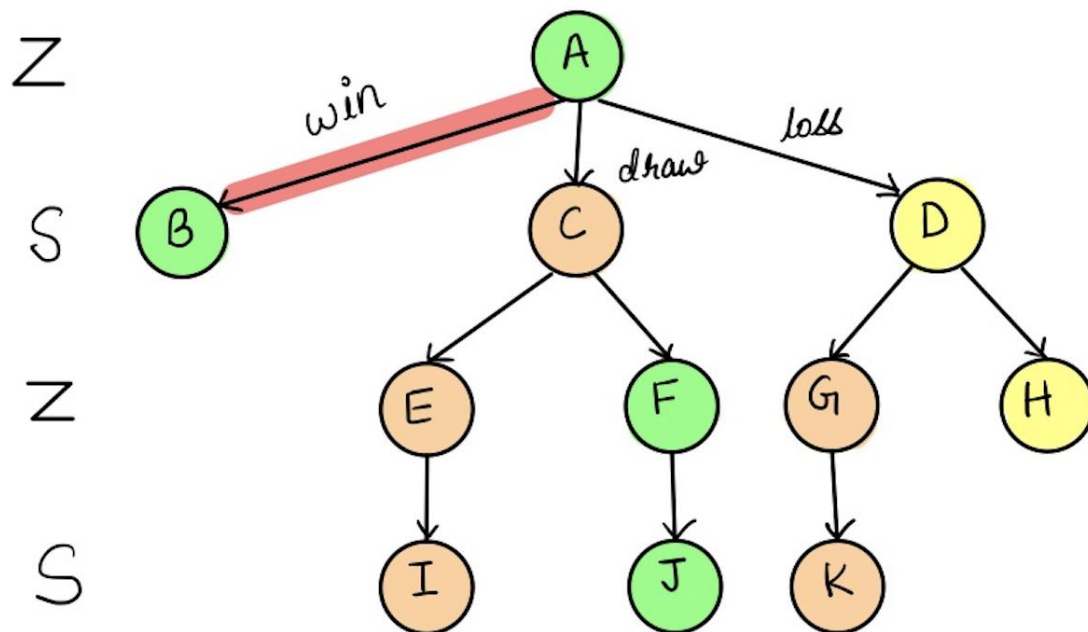

5. Explanation of algorithm with an example:

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;
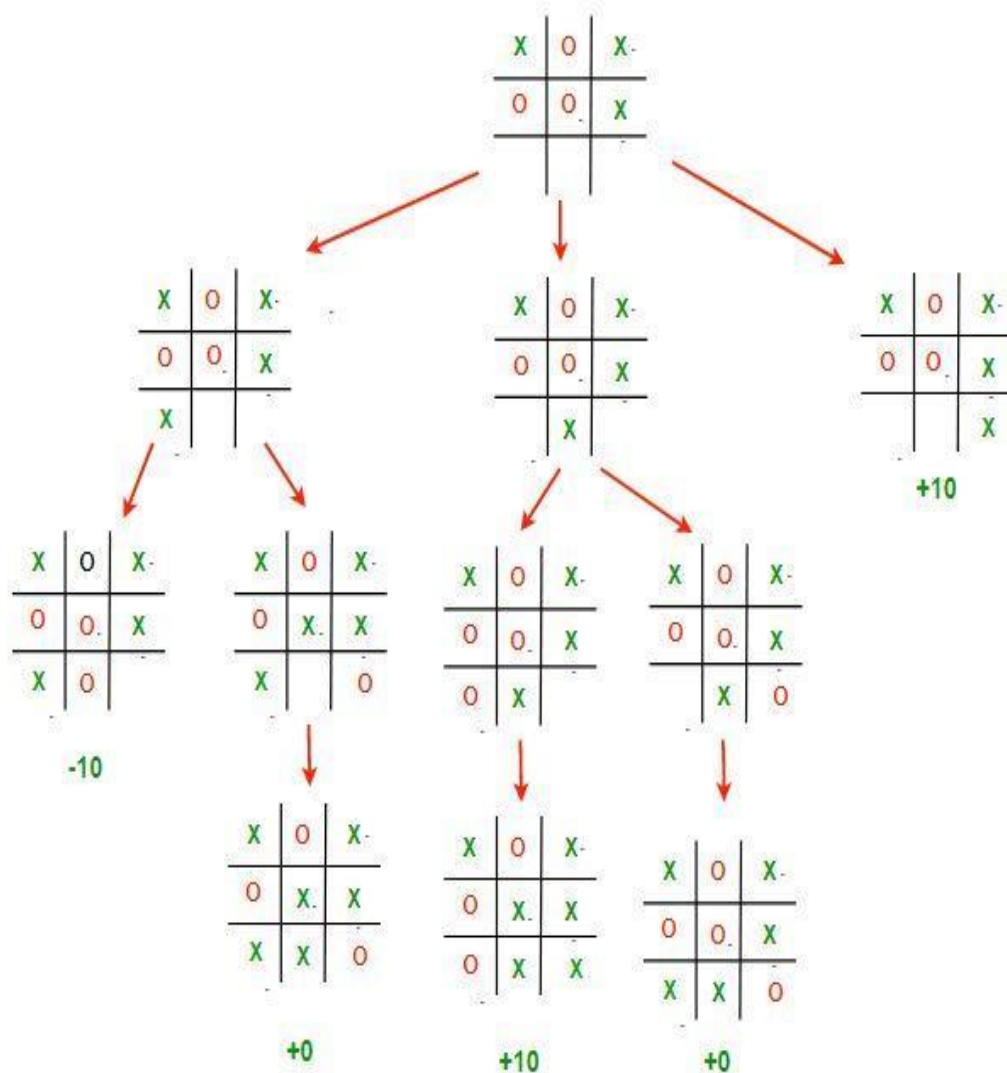
Remember, even though X has a possibility of winning if he plays the middle move, O will never let that happen and will choose to draw instead.

Therefore the best choice for X is to play [2,2], which will guarantee a victory for him... Minimax may confuse programmers as it thinks several moves in advance and is very hard to debug at times. Remember this implementation of the minimax algorithm can be applied to 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 up to a certain depth and use the evaluation function to calculate the value of the board.

6. Code and Output:

```python
# Python3 program to find the next optimal move for a player
player, opponent = 'x', 'o'


# This function returns true if there are moves
# remaining on the board. It returns false if
# there are no moves left to play.
def isMovesLeft(board) :
    for i in range(3) :
        for j in range(3) :
            if (board[i][j] == '_') :
                return True
    return False


# This is the evaluation function as discussed
def evaluate(b) :
# Checking for Rows for X or O victory.
    for row in range(3) :
        if (b[row][0] == b[row][1] and b[row][1] == b[row][2]) :
            if (b[row][0] == player) :
                return 10
            elif (b[row][0] == opponent) :
                return -10


    # Checking for Columns for X or O victory.
    for col in range(3) :
        if (b[0][col] == b[1][col] and b[1][col] == b[2][col]) :
            if (b[0][col] == player) :
                return 10
```

```python
        elif (b[0][col] == opponent) :
            return -10

    # Checking for Diagonals for X or O victory.
    if (b[0][0] == b[1][1] and b[1][1] == b[2][2]) :
        if (b[0][0] == player) :
            return 10
        elif (b[0][0] == opponent) :
            return -10

    if (b[0][2] == b[1][1] and b[1][1] == b[2][0]) :
        if (b[0][2] == player) :
            return 10
        elif (b[0][2] == opponent) :
            return -10

    # Else if none of them has 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
def minimax(board, depth, isMax) :
    score = evaluate(board)

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

```python
# 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) :
    best = -1000

    # Traverse all cells
    for i in range(3) :
        for j in range(3) :

            # 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,
                                not isMax) )
                # Undo the move
                board[i][j] = '_'
```

```python
            return best

    # If this minimizer's move
    else :
        best = 1000

        # Traverse all cells
        for i in range(3) :
            for j in range(3) :

                # 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, not isMax))

                    # Undo the move
                    board[i][j] = '_'
        return best

# This will return the best possible move for the player
def findBestMove(board) :
    bestVal = -1000
    bestMove = (-1, -1)

    # Traverse all cells, evaluate minimax function for
    # all empty cells. And return the cell with optimal
    # value.
    for i in range(3) :
        for j in range(3) :

            # Check if cell is empty
            if (board[i][j] == '_') :

                # Make the move
```

```python
            board[i][j] = player


            # compute evaluation function for this move.
            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 = (i, j)
                bestVal = moveVal
    print("The value of the best Move is :", bestVal)
    print()
    return bestMove
# Driver code
board = [
    [ 'x', 'o', 'x' ],
    [ 'o', 'o', 'x' ],
    [ '_', '_', '_' ]
]
bestMove = findBestMove(board)
print("The Optimal Move is :")
print("ROW:", bestMove[0], " COL:", bestMove[1])
```

OUTPUT:

7. Complexity analysis:

Time complexity:

As it performs DFS for the game tree, the time complexity of the Min-Max algorithm is O(bm), where b is the branching factor of the game tree, and m is the maximum depth of the tree.

Space Complexity:

The space complexity of the Mini-max algorithm is also similar to DFS which is O(bm).

8. Conclusion:

Minimax may confuse programmers as it thinks several moves in advance and is very hard to debug at times. Remember this implementation of the minimax algorithm can be applied to 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 up to a certain depth and use the evaluation function to calculate the value of the board.

The testing results have revealed that the performance of the pencil-and-paper algorithm is better than the Backtracking algorithm with respect to the computing time to solve any puzzle.

The Backtracking algorithm seems to be a useful method to solve any Sudoku puzzles and it can guarantee finding at least one solution. However, this algorithm is not efficient because the level of difficulties is irrelevant to the algorithm. In other words, the algorithm does not adopt intelligent strategies to solve the puzzles. This algorithm checks all possible solutions to the puzzle until a valid solution is found which is a time-consuming procedure resulting in an inefficient solver. As it has already been stated the main advantage of using the algorithm is the ability to solve any puzzles and a solution is certainly guaranteed.

9. References

[1] Knuth, Donald (1997). Fundamental Algorithms, Third Edition, ISBN 0-201-89683-4. [2] Ch. X u, W. X u, 2009, The model and algorithm to Estimate The Difficulty Levels of Sudoku puzzles, Journals of Mathematics Research.. [3] Lewis, R Metaheuristics, 2007, Can solve Sudoku puzzles. Journal of heuristics,13(4),387-401. [4] Xu J, 2009, Using backtracking method to solve Sudoku puzzle, computer programming skills & maintenance 5, pp 17-21. [5] Chakraborty,r, Palidhi, S Banerjee, 2014, An optimized Algorithm for solving combinatorial problems using reference graph IOSR Journal of CS 16(3PP1-7).