Into the Depths of Game Trees: A Minimax Algorithm Analysis

Blix D. Foryasen
National University
Manila, Philippines
foryasenbd@students.national-u.edu.ph

***Abstract*— Commonly used in decision-making and game theory, the minimax algorithm is a decision-making algorithm to find the most feasible outcome for a player to take. Two players assume that their opponent plays optimally to maximize their wins while minimizing their losses. Game trees represent the possible actions performed alternatively by the players to reach a definite conclusion. The algorithm analyzes through a depth-first search, starting from the initial game state or root node down to the terminal nodes before backtracking and returning the best possible terminal state a player needs to take. All nodes and respective child nodes are recursively traversed for every level of the game tree, first branching out from the initial game state. As the algorithm goes deeper into the game tree, the search space and computational complexity increase exponentially. Considering the algorithm's exponential growth, optimizations should be considered to make the algorithm feasible for complex games and tasks.**

Keywords—game tree search, decision-making, backtracking, minimax algorithm, two-player zero-sum games,

I. Algorithm Description

The minimax algorithm is a recursive backtracking algorithm that evaluates the best possible action for a player with the assumption that their opponent plays optimally [1]. Such a decision rule provides the player the strategy to minimize the potential loss while maximizing their wins [2], [3], hence the name "minimax." Since initially proven in 1928 by mathematician John von Neumann [4], the minimax logic remains a foundational component used in Artificial Intelligence (AI), decision theory, and game theory, constantly optimized and modified in developing futuristic decision-making models and engines [5]. One such breakthrough happened in 1997 when World Chess Champion Garry Kasparov lost to IBM's supercomputer Deep Blue in an official match, 3.5-2.5 [6]. Ever since then, the algorithm has been constantly optimized to remain feasible for future applications and developments.

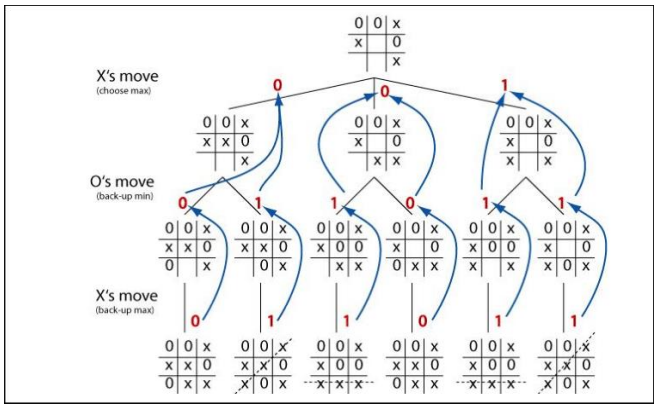

Fig. 1. Visualized step-by-step process of minimax algorithm.

Game trees serve as the foundation for the minimax algorithm to provide the most optimal decision by a player [7]. Given the player to move, the root node, signifying the current game state or position, and a starting depth, all possible successive game states are analyzed for every level. Edges represent a move done by a player to reach another node or game state. The tree branches out, simulating possible moves that happen until an outcome appears, and eventually increasing the search depth of the game tree. Often represented by a terminal node or leaf node, outcomes have a respective evaluation value. Backtracking returns a node value based on the minimax rule. The evaluation repeats until it reaches the root node and returns the best move for the current player. In a complete and finite search tree, the algorithm returns a solution if there exists one.

II. Formal Algorithm

The algorithm performs recursive exploration and backtracking starting from a current game state, a starting search depth, and the maximizing or minimizing player [1]-[7]. Fig. 2 represents the algorithm's pseudocode, which takes the two inputs to return the most optimal node value or game state the player needs to take next.

```
function minimax(game_state, depth, is_maximizing)
    if current instance of the game_state is a terminal state:
        return value of the terminal state

    if is_maximizing
        max_value = -∞
        for each move in all possible moves
            score = minimax(game_state, depth + 1, False)
            max_value = max(score, max_value)
        return max_value
    else
        min_value = ∞
        for each move in all possible moves
            score = minimax(game_state, depth + 1, True)
            min_value = min(score, min_value)
        return min_value
```

Fig. 2. Pseudocode of recursive minimax algorithm checking all possible outcomes

Starting from the root node, the algorithm recursively traverses the tree until it reaches the base case or the terminal node. For every level of the game tree, the player in play recursively alternates between the minimizer and maximizer, increasing the search depth. Once the terminal node's value is returned, optimal values are updated based on the player in play and the minimax rule for every backtracking step in the recursive stack. The entire process repeats until all values have been visited and analyzed. Upon reaching the root node after all the backtracking steps, the algorithm provides the player-in-play returns with which evaluated value to consider. The maximizer player chooses the maximum possible score, while the minimizer tries to get the minimum possible score.

III. Implementation and Annotation

```
def minimax(board, depth, is_maximizing):
    winner = check_winner(board)
    if winner is not None:
        return winner_score(winner)

    if is_maximizing:
        best_score = float('-inf')
        for move in get_possible_moves(board):
            board[move] = 'X'
            score = minimax(board, depth + 1, False)
            board[move] = ' '
            best_score = max(score, best_score)
        return best_score

    else:
        best_score = float('inf')
        for move in get_possible_moves(board):
            board[move] = 'O'
            score = minimax(board, depth + 1, True)
            board[move] = ' '
            best_score = min(score, best_score)
        return best_score
```

Base case: return evaluated node value O(1)

Explore possible moves O(b)

Recursive step: Increase search depth O(b^d)

Update best value O(1)

Return best value O(1)

Final time complexity: O(b^d)

Fig. 3. Annotated Python implementation of Minimax algorithm in Tic-Tac-Toe

Fig. 3 shows the Python implementation of the Minimax algorithm as a function in a 3x3 Tic-Tac-Toe game that implements recursive exploration of trees through depth-first search and evaluation using backtracking [3], [7].

All possible moves are considered, branching out from the current game state or the root node of the game tree down to the possible terminal outcomes of the game. Every subtree, except for the terminal states, has children that need to be explored individually.

The recursion call is the exploratory process for all children per level of the game tree. The deeper the game tree, the more children per level, and a combination of both all increases the instance of the recursive step. For every recursive step, the roles of maximizing and minimizing players' swaps signify that the players alternatively performed a particular move to get to the new game state.

The recursive step stops once the terminal states are reached, and the evaluated value is returned. Values of terminal nodes denote whether the game resulted in a tie or a player winning or losing following a series of game states.

Upon evaluating terminal states, the algorithm backtracks to the previous game state and compares the returned value to the existing heuristic value. For all non-terminating nodes, the Maximizing and Minimizing nodes' optimal values are initialized to negative and positive infinity, respectively.

Each backtracking step updates the best value for every recursive stack until it reaches the current state. The algorithm finally returns the most feasible solution for the current player, whether maximizing or minimizing.

IV. Complexity Analysis

The Minimax algorithm visits and evaluates each game state in the game tree to determine which move is the most feasible for the current player. Visiting and assessing game states and comparing and updating the best values for each player involves a time complexity of O(1). However, to better visualize the algorithm's time complexity, the depth (d) of the search space or tree and the branching factor (b), the average number of child nodes of each node, should be considered.

The root node has a 'b' number of children, each having 'b' children of their own. Since all game states are explored by the minimax algorithm in the game tree of depth 'd', the total number of evaluated nodes is:

$$T(b) = 1 + b + b^2 + b^3 + \ldots + b^d \qquad (1)$$

Equation (1) shows that for every depth of the game tree, there exists b^d nodes to be explored by the algorithm. Thus, the time complexity of the minimax algorithm is O(b^d). Given its nature of exploring all nodes in the game tree, the average, best, and worst-case time complexity of the algorithm remains to be O(b^d).

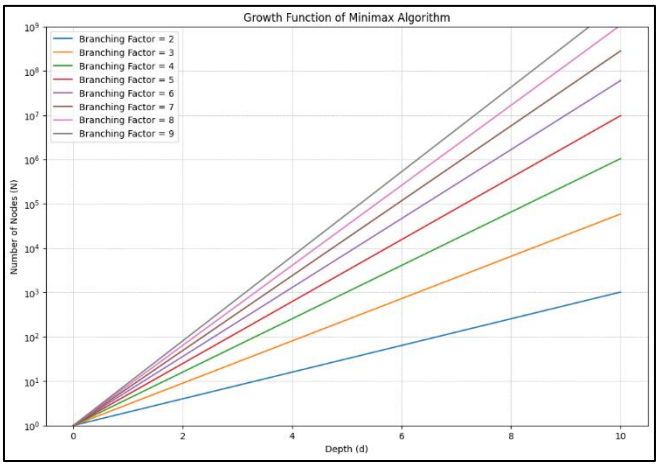

Fig. 4. Frequency analysis of evaluated nodes of Minimax Algorithm based on branching Factor and search depth

Fig. 4 shows the growth function of the minimax algorithm for varying branching factors. The x-axis represents the search depth of the algorithm while the y-axis logarithmically indicates the number of evaluated nodes. As observed, expanding the search depth, the branching factor of the tree, or both leads to increasing search spaces.

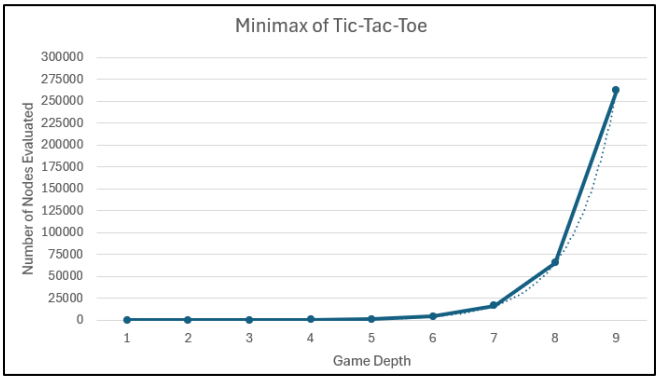

Fig. 5. Frequency analysis of Minimax Algorithm when applied into a game of Tic-Tac-Toe

Fig. 5 shows the application of minimax in a Tic-Tac-Toe game tree starting from the initial game state, that is, an empty 3x3 board. As the search depth increases, exponential growth is observed in computational complexity. The exponential nature of the search space is due to the algorithm's recursive evaluations. Applying the algorithm in large game trees could result in substantial computational constraints, leading to slower, less efficient results.

REFERENCES

[1] R. Nasa, R. Didwania, S. Maji and V. Kumar, "Alpha-beta pruning in mini-max algorithm – an optimized approach for a connect-4 game," International Research Journal of Engineering and Technology, vol. 5, no. 4, pp. 1637 - 1641, April 2018.

[2] S. S. Shevtekar, M. Malpe and M. Bhaila, "Analysis of game tree search algorithms using minimax algorithm and alpha-beta pruning," International Journal of Scientific Research in Computer Science Engineering and Information Technology, vol. 8, no. 6, pp. 328-333, 28 November 2022.

[3] S. H. Alkaraz, E. El-Seidy and N. S. Morcos, "Tic-tac-toe: understanding the minimax algorithm," Journal of Game Theory, vol. 9, no. 1, pp. 1-7, 2020.

[4] X. Kang, Y. Wang and Y. Hu, "Research on different heuristics for minimax algorithm insight from connect-4 game," Journal of Intelligent Learning Systems and Applications, vol. 11, no. 2, pp. 15-31, May 2019.

[5] S. Videgain and P. G. Sanchez, "Performance study of minimax and reinforcement learning agents playing the turn-based game Iwoki," Applied Artificial Intelligence, vol. 35, no. 10, 15 June 2021.

[6] J. Madake, C. Deotale, G. Charde and S. Bhatlawande, "CHESS AI: machine learning and minimax based chess Engine," in 2023 International Conference for Advancement in Technology (ICONAT), 2023.

[7] A. W. Toure, "Evaluation of the use of minimax search in connect-4," Applied Mathematics, vol. 14, no. 6, pp. 419-427, 13 June 2023