Experiment 4
Game Playing Agent- Minimax- Alpha-Beta Pruning

Jagruti Piprade
Computer Science Engineering
202151067@iiitvadodara.ac.in

Makwana Harsh Maheshkumar
Computer Science Engineering
202151082@iiitvadodara.ac.in

Khushi Saxena
Computer Science Engineering
202151078@iiitvadodara.ac.in

Nitin Gautam
Computer Science Engineering
202151101@iiitvadodara.ac.in

Google collab notebook for this assignment can be found here.

***Abstract*—This research paper aims to examine how computers play games using smart strategies. We study how strategies like Minimax and alpha-beta pruning improve computer gameplay in games like Tic-Tac-Toe and Nim. These strategies help computers make smarter decisions by cutting down unnecessary choices. This report helps us understand how computers can play games really well.**

I. Learning Objective

The aim of this experiment is to see how using Minimax and alpha-beta pruning makes computer gameplay better. We want to find out if these methods help computers play games smarter and faster, especially in situations where they have to compete against an opponent.

II. Introduction

Computers and games have always fascinated us, especially when computers can play games just like humans. We introduce some key concepts of game playing and decision-making algorithms.

In this experiment, we cover a few important ideas. First is the Playing Agent, which can be either a person or a computer in a game who makes choices. Then, we talk about two key strategies: Minimax and Alpha-Beta Pruning. Minimax is like a smart decision-making process for games, especially to avoid losing badly. Alpha-Beta Pruning is a trick to make Minimax work faster by skipping some unnecessary steps. These concepts are the building blocks for making computer gameplay better.

III. Problem I: Noughts and Crosses (Tic-Tac-Toe)

Problem Statement: What is the size of the game tree for Noughts and Crosses? Sketch the game tree.

Tic-Tac-Toe is a two-player game played on a 3x3 grid, where players take turns marking spaces with their respective symbols, usually X and O. The objective is to form a line of three of your symbols horizontally, vertically, or diagonally. The game keeps going until one player gets three in a row and wins, or until the grid is full with no winner, resulting in a draw. If both players make the best moves they can, the game will always end in a draw.

To calculate the size of the game tree for Noughts and Crosses, we need to consider two main factors: branching factor and depth.

- **Branching Factor:** The branching factor represents the number of available moves for each player at any given point. In a 3x3 grid, there are initially 9 empty cells, so the branching factor is 9 for the first move. As the game progresses, the number of empty cells decreases, leading to a decreasing branching factor.
- **Depth:** Depth refers to how many moves are made in a game until it ends. In the worst-case scenario, where the game reaches a draw, each player will make up to 9 moves, completely filling the grid. Thus, the maximum depth of the game tree can reach 9 levels.

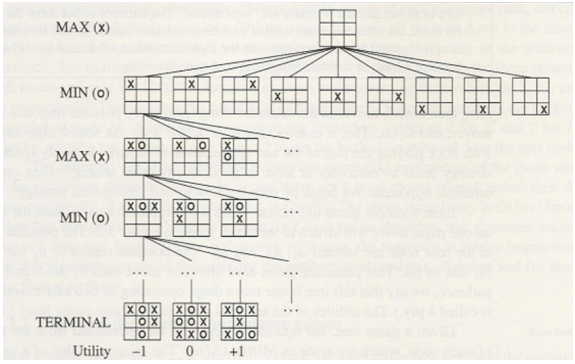

Fig. 1. Game tree for noughts and crosses

S_0 represents the initial state of a game of Tic-Tac-Toe.

$$S_0 = \begin{bmatrix} - & - & - \\ - & - & - \\ - & - & - \end{bmatrix}$$

Each level represents a player's turn. Nodes depict the game state after a player's move, starting from the initial empty grid at the root node. Branches represent possible moves, expanding exponentially with each level. Leaf nodes signify terminal states, such as win, lose, or draw.

The game tree initially has 9! (362,880) edges, which includes games that extend beyond a victory. Removing these games leaves 255,168 edges. This number might appear unexpectedly large for a simple game like Noughts and Crosses due to several reasons.

1) There's only one starting position.
2) Each player has 9 possible moves in the first turn and 8 in subsequent turns because one square is already occupied.
3) Players taking turns to make moves effectively doubles the number of possible positions at each level of the game tree.

When we combine similar game boards, we end up with 26,830 edges, mostly because we're dealing with rotations and reflections. But this count is too high because it includes some illegal positions and doesn't consider symmetrical scenarios, like flipping the board. In reality, there are fewer unique positions than this count suggests.

IV. Problem II: The Nim Game

Nim is a two-player game involving heaps of objects, such as stones, coins, or matchsticks. Players take turns removing objects from one or more heaps according to set rules. Typically, a player can take any number of objects from a single heap during their turn. The game continues until no objects remain. The player who takes the last object wins.

Algorithm

1) Define Initial State: Begin with the initial configuration of the game, specifying the number of objects in each pile.
2) Build Game Tree: Construct a game tree showing every possible move and counter-move from the game's start. Each point on the tree represents a game situation, and each line between points shows a possible move from one situation to another.
3) Assign Terminal Values: Assign terminal values to the leaf nodes of the tree based on whether the state is a win, loss, or draw for player-2. In Nim, a state where player-2 has no legal move left signifies a win, while a state where player-1 has no legal move left results in a loss.

Fig. 2. Nim game initial State

Minimax

1) Start at the root node of the game tree.
2) For each node:
   - Assign the terminal value to the node if it is a leaf node.
   - If the node represents a MAX node (indicating player 1's turn), select the child node with the highest value, which represents player 1's best move.
   - If the node represents a MIN node (indicating player 2's turn), choose the child node with the minimum value, reflecting player 2's optimal move.
   - Update the values up the tree until reaching the root node.

Decision making: Evaluate the computed values to determine the best move for player 1. Since player 2 always seeks to maximize their chances of winning, player 1's optimal move is determined by selecting the move that minimizes the maximum potential loss for player 2.

Game Outcome: Regardless of player 1's move, player 2 responds optimally to uphold their winning strategy. This ensures that player 2 will always have a winning response, resulting in a victory for player 2.

This algorithm shows how the MINIMAX method can be used to analyze Nim, revealing that player 2 will always win, no matter what player 1 does. It's because player 2 can always make the best move to stay ahead.

V. Problem III: Alpha beta pruning

Alpha-beta pruning is an optimization technique used in game trees to reduce the number of nodes that need to be evaluated. It's based on the Minimax algorithm, which aims to find the best move for a player assuming the opponent also plays optimally.

- **Alpha:** The alpha value represents the best value that the maximizing player (e.g., player 1) can achieve so far

along the path of any of the possible moves from this state.

- **Beta:** The beta value represents the best value that the minimizing player (e.g., player 2) can achieve so far along the path of any of the possible moves from this state.

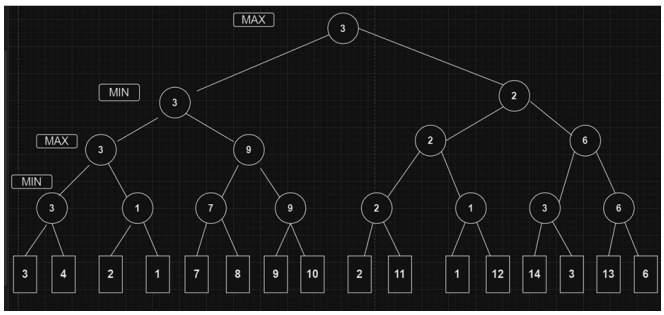

Fig. 3. Minimax tree

The code we've implemented uses a Tic-Tac-Toe board where rows and columns start counting from 0. So, in this setup, the top-left corner of the board is represented by the coordinates (0, 0). As a result, the algorithm suggests placing an X in the top-left corner as the best move after analyzing the game situation.

It uses the MINIMAX algorithm with alpha-beta pruning to find the best move for player X in Tic-Tac-Toe. It looks at all possible future game scenarios up to a certain depth, ensuring the best decisions are made.

VI. Problem IV: Time Complexity

When using alpha-beta pruning with perfectly ordered leaf nodes, the algorithm visits nodes in a depth-first manner, exploring branches of the game tree until it reaches a terminal state or a cutoff condition. In the best-case scenario, where leaf nodes are perfectly ordered, the algorithm can prune entire subtrees without fully exploring them, significantly reducing the time complexity.

Let's define the recurrence relation:

N(d): Number of nodes visited at depth

b: Effective branching factor (average number of children per node).

m: Depth of the tree

To simplify, let's assume that the effective branching factor b remains constant at each level of the tree.

Recurrence relation: At each level d, the algorithm explores at most b children (assuming perfect ordering). However, due to pruning, the algorithm may explore fewer nodes depending on the values of alpha and beta.
Therefore,

$$N(d) = b \cdot N(d-1)$$

The number of nodes explored at depth d is directly related to the number of nodes explored at depth $d-1$, multiplied by the effective branching factor b.
Solving the relation we get,

$$N(d) = b^d$$

The total visited nodes throughout the search is the sum of nodes explored at each depth from 1 to m.

$$\sum_{d=1}^{m} N(d) = \sum_{d=1}^{m} b^d = \sum_{d=1}^{m} \frac{b^{m+1} - b}{b-1}$$

Therefore, the time complexity of alpha-beta pruning under perfect ordering of leaf nodes is O(b^m)

Considering the depth m halved due to alpha-beta pruning, the time complexity becomes $O(b^{m/2})$, where b is the branching factor.

VII. Conclusion

We closely studied the Noughts and Crosses game, understanding its structure, rules, and the different tactics players can use to outsmart their opponents and secure victory. We also explained the game of Nim, showing that no matter what Player-1 does, Player-2 has a better chance of winning by thinking carefully about the game and making the best moves possible. We implemented two fundamental algorithms, MINIMAX and alpha-beta pruning, to automate decision-making processes in Noughts and Crosses. Alpha-beta pruning helps make calculations faster by cutting down on unnecessary work, especially when game states are ordered optimally.

References

[1] https://en.wikipedia.org/wiki/Game$_complexity$
[2] https://www.cs.swarthmore.edu/ mitchell/classes/cs63/s24/labs/03.html
[3] https://medium.com/chat-gpt-now-writes-all-my-articles/advanced-ai-alpha-beta-pruning-in-python-through-tic-tac-toe-70bb0b15db05