Addis Ababa University

School of Information Technology and Engineering 2024

Master of Science in Artificial Intelligence

Design and Implementation of a Minimax Game-Playing Agent

Instructor: Dr. Natnael Argaw Wondimu
By Fransi Ayele, ID: GSE/1254/18

1. Introduction

Adversarial search is a cornerstone of Artificial Intelligence, modeling decision-making processes in competitive environments where agents play against an opponent with conflicting goals. The objective of this project was to design and implement an intelligent agent capable of playing a deterministic, zero-sum, perfect-information game optimally.

For this assignment, **Tic-Tac-Toe** was selected as the domain. While the game space is relatively small (approx. 3^9 states), it serves as an ideal testbed for demonstrating the correctness of the **Minimax Algorithm** and the efficiency gains provided by **Alpha-Beta Pruning**. The implemented agent does not merely play to win; it utilizes a depth-scaled heuristic to distinguish between a "fast win" and a "slow win," ensuring the most efficient path to victory is chosen.

2. Formal Game Representation

In accordance with the assignment requirements, the game is modeled as a state-space search problem defined by the tuple (S, S0, A, T, U):

- **States (S):** The state is represented by a linear array of size 9, board[0...8], where each index corresponds to a cell on the 3x3 grid. Each cell c is either EMPTY, 'X', or 'O'.
- **Initial State (S0):** An empty array where all cells are marked as EMPTY.
- **Actions (A):** The set of legal moves defined by finding all indices 'i' where board[i] is EMPTY.
- **Transition Model (T):** The function Result (s, a) returns a new state s' where the player's marker is placed at index a.
- **Utility Function (U):** The terminal states are evaluated numerically from the perspective of the Maximizing Agent ('X'):
    - ➢ **Win (X):** Positive Utility (+10)
    - ➢ **Loss (X):** Negative Utility (-10)
    - ➢ **Draw:** Zero Utility (0)

3. Algorithm Implementation

3.1 The Minimax Decision Rule

The core intelligence is built upon the Minimax algorithm. The agent (Maximizer) recursively simulates all possible future moves to the terminal state. It operates on the assumption that the opponent (Minimizer) will always play optimally to minimize the agent's score.

The implementation utilizes **Backtracking**. Rather than creating deep copies of the board object for every simulation (which is memory-intensive), the algorithm applies a move, recursively calls the function, and then immediately invokes an undo_move function to restore the board state.

3.2 Alpha-Beta Pruning

To address the exponential time complexity of standard Minimax (O(b^d)), Alpha-Beta pruning was implemented. This optimization maintains two values during the search:

- **Alpha:** The best value the Maximizer can guarantee so far.
- **Beta:** The best value the Minimizer can guarantee so far.

If at any node the algorithm discovers a move that is worse than a previously examined move (i.e., Beta <= Alpha), that branch is "pruned" (discarded), as it will not influence the final decision. This significantly reduces the number of nodes expanded.

3.3 Depth-Scaled Heuristic ("Smart" Logic)

A standard Minimax agent treats a win in 1 move as equal to a win in 5 moves. To make the agent "smarter" and more aggressive, a depth penalty was introduced into the utility function:

- **If X wins:** Score = 10 - depth
- **If O wins:** Score = depth - 10
- **If Draw:** Score = 0

This modification ensures that the agent prefers winning at shallower depths (fast wins). Conversely, if a loss is inevitable, the agent attempts to prolong the game (maximizing the negative score closer to 0) in hopes the human player makes an error.

4. Design Choices and Architecture

4.1 Object-Oriented Design (OOP)

To ensure code modularity and maintainability, the system was split into two distinct classes:

1. TicTacToeGame: This class acts as the "Physics Engine." It handles board rendering, move validation, and win-condition checking. It has no knowledge of AI strategy.
2. MinimaxAgent: This class encapsulates intelligence. It contains recursive algorithms and evaluation logic.

This **Separation of Concerns** adheres to software engineering best practices, making the code easier to debug and extend.

4.2 Opening Optimization

Running a full Minimax search on an empty board is computationally expensive (9 factorial checks). A heuristic optimization was added: if the board is empty, the agent immediately occupies the **Center (Index 4)**. This is theoretically the strongest opening move in Tic-Tac-Toe and eliminates the initial calculation delay.

4.3 Non-Deterministic Behavior

To prevent the agent from feeling "robotic" (playing the exact same game every time), the agent collects *all* moves that result in the optimal score and selects one randomly. This maintains optimality while adding variety to the gameplay.

5. Challenges and Solutions

- **State Management (Backtracking):**
  - ➢ *Challenge:* Initially, the recursive calls were permanently modifying the board, causing the AI to hallucinate moves that hadn't happened.
  - ➢ *Solution:* A strict sequence of make_move -> recurse -> undo_move was enforced to ensure the board state remained consistent at every level of the recursion tree.
- **Handling Terminal States:**
  - o *Challenge:* Detecting a draw condition correctly.
  - o *Solution:* The is_game_over check was ordered strictly: Check Win first, then Check Draw. This prevents a board that is full (but has a winner on the last move) from being misclassified as a draw.

6. Conclusion

The implemented agent successfully meets all assignment objectives. Through the use of Minimax with Alpha-Beta pruning, the agent plays optimally and is mathematically unbeatable. The addition of depth-scaled heuristics transforms the agent from a passive player into an aggressive one that seeks the quickest victory. The project demonstrated the power of recursive search algorithms in solving zero-sum games and the importance of heuristic optimizations in reducing computational overhead.