University of North Carolina at Chapel Hill

Department of Computer Science

Tic Tac Toe: Optimal Agent using Minimax Algorithm and Alpha-Beta Pruning

Jean Cheng
Kathleen Fort
Nathan Jacobs
Aaditi Jadhav
Arun Somasundaram

COMP 560 Section 02
Spring 2025

Contents

1 Abstract

This project focuses on developing an intelligent agent that plays Tic Tac Toe optimally. Using the minimax algorithm with alpha-beta pruning, the agent evaluates potential moves to maximize its winning chances while minimizing the opponent's opportunities. This project was implemented in Python using Visual Studio Code. The agent successfully plays optimally, ensuring a win or draw depending on the opponent's skill.

2 Introduction and Motivation

Tic Tac Toe is a classic two-player game that has served as a benchmark for developing and evaluating decision-making algorithms in artificial intelligence. Although it is simple, it provides an ideal environment for exploring concepts such as adversarial search and optimal play strategies.

In this project, we implemented an agent that can intelligently play Tic Tac Toe using the Minimax algorithm optimized with alpha-beta pruning. The use of Minimax enables the agent to evaluate future game states under the assumption that both sides will play perfectly, while using alpha-beta pruning to significantly reduce the number of states evaluated by eliminating suboptimal branches.

The agent consistently plays optimally and will never lose to an opponent who does not play perfectly. If the opponent does play perfectly, the game will end up in a draw. The agent's performance validates the correctness and efficiency of our approach and showcases how widespread AI adversarial searches can be effectively applied to smaller games.

3 Related Works

The minimax algorithm and its optimizations have long been central to research in adversarial game-playing agents. The original concept stems from von Neumann's minimax theorem von Neumann (1928), which laid the foundation for decision-making under adversarial conditions. In the context of game trees, Knuth and Moore (1975) formally introduced alpha-beta pruning as a method to reduce the number of nodes evaluated in a minimax search without altering the final decision, improving efficiency from $O(bd)$ to $O(bd/2)$ in the best case, where b is the branching factor and d is the depth of the tree.

Although Tic Tac Toe is a fully solvable game with a small state space, it remains a classic example for evaluating exact search algorithms. Slagle and Dixon (1969) demonstrated early applications of machine reasoning to games like Tic Tac Toe and symbol manipulation. Their work highlighted how rule-based systems and tree search could be combined to produce intelligent behavior in constrained domains.

More recent work explores extensions of minimax to stochastic and imperfect information games, but deterministic games such as Tic Tac Toe serve as useful benchmarks for implementing and analyzing classical methods. For instance, Barr and Feigenbaum (1981) investigated efficient search control strategies for game trees, with implications for small board games like Tic Tac Toe. Additionally, Knuth and Moore (1975) analyzed the performance of alpha-beta pruning across various game tree structures, demonstrating its efficiency in reducing the number of nodes evaluated during search.

This project builds upon these foundational ideas by implementing a minimax agent with alpha-beta pruning for standard 3x3 Tic Tac Toe, serving as a demonstration of classical search strategies in a controlled, deterministic setting.

4 Methods

The system is built in Python to handle the game logic and AI computations. The implementation of the agent involves the following key components:

4.1 Minimax Algorithm

The core of our Tic Tac Toe agent's decision-making process is the Minimax algorithm, which recursively evaluates all possible moves in the game tree. The algorithm assumes that both the agent and the opponent play optimally, meaning that the agent will choose moves that maximize its own score while minimizing the opponent's score. To implement this, we define a recursive function `minimaxSearch` that explores every possible game state by evaluating the outcome of making each available move. For each potential move, the function assigns a score based on the game's outcome:

- X (agent) wins: +10
- O (opponent) wins: -10
- Tie: 0

The function proceeds by checking all possible moves, updating the board, and recursively calling `minimaxSearch` on the new game state. The score for each move is determined by whether it leads to a win, loss, or tie. Once all moves are evaluated, the best move (with the highest score for the agent or the lowest score for the opponent) is selected.

4.2 Alpha-Beta Pruning

To optimize the Minimax algorithm's performance, we implement Alpha-Beta pruning, which reduces the number of nodes evaluated in the search tree. This method works by maintaining two values—alpha and beta—that represent the minimum score the maximizing player is assured of (alpha) and the maximum score the minimizing player is assured of (beta). As the search progresses, branches of the tree that cannot influence the final decision are pruned, resulting in significant computational savings.

In our implementation, the function `minimaxSearch` uses alpha and beta parameters to prune unnecessary branches. If the current branch cannot improve upon the best found solution so far (alpha for the maximizing player or beta for the minimizing player), the function terminates early, avoiding the exploration of those branches.

4.3 Game Play

4.3.1 Game State Representation

The game state is represented by a 3x3 grid, where each cell can be empty, marked with an X (the agent's symbol), or marked with an O (the opponent's symbol). The agent plays as X and the opponent plays as O. Each possible state of the game is encapsulated in a search node, which contains the following attributes:

- Current game state: A 3x3 grid representing the state of the game.
- Parent node: The node from which the current node is derived.

- Possible next moves: A list of possible moves that the player can make in the current games state.
- Current depth: The current depth of the node in the search tree.

4.3.2 Successor Function

The successor function generates all possible successor states from the current game state. Each successor represents a possible move that the agent or the opponent could make. The function ensures that only valid moves (i.e., moves made in blank spaces) are considered, and each resulting game state is evaluated based on its strategic value. Specifically, the successor function looks for moves that lead to:

- A win: If the agent's move creates three of its symbols in a row.
- Blocking the opponent: If the opponent is about to win and the agent must block them.
- Strategic advantage: Moves that lead to favorable positions or potential winning setups.

4.3.3 User Interaction

The user interacts with the game via the graphical user interface, which displays the Tic Tac Toe grid. The player can choose whether they want to play first or second. If the player is second, the agent makes its move immediately after the player.

The game alternates between the agent and the opponent, and the state of the game is updated after each move. The game ends when there is a winner or a tie. The result is displayed on the screen, indicating whether the agent or the opponent won, or if the game ended in a tie.

4.4 Game Simulations

To evaluate our agent's performance, we developed a simulation framework that can automate games against two types of opponents. The first type is an imperfect opponent that selects moves randomly, simulating suboptimal human play. The second type is an optimal opponent that uses the same minimax algorithm with alpha-beta pruning, ensuring the opponent plays perfectly. In each simulation round, the agent and opponent alternate moves until the game ends in a win or tie. We also implemented a batch simulation function that plays multiple rounds and records the number of wins, losses, and ties for analysis.

5 Results

Our implementation resulted in a fully functional Tic Tac Toe game utilizing the minimax algorithm with alpha-beta pruning.

To evaluate the agent's performance, we conducted simulations using the `nSimulations` function, where a specified number of games `n` were played against either a random or an optimal opponent. For example, when `n=500`, simulations against a random opponent showed that the agent (playing as X) never lost, with outcomes consisting exclusively of either wins for the agent or ties.

Against an optimal opponent, where the opponent also employed the minimax algorithm, all games ended in ties, consistent with the theoretical expectation for perfect Tic Tac Toe play. These results validate the correctness and effectiveness of our minimax-based decision-making.

In addition to confirming the agent's strategic performance, the integration of alpha-beta pruning improved the efficiency of the minimax search. Alpha-beta pruning reduced the number of nodes evaluated by eliminating branches that could not influence the final decision, thereby accelerating the move-selection process. Although pruning does not affect the outcome of games, it optimizes runtime, which is especially significant when running a large amount of game simulations. In larger or more complex games with higher branching factors, such improvements in search efficiency would become even more critical.

6 Conclusion

Throughout our work in this project, we created an intelligent Tic Tac Toe agent that plays optimally using the minimax algorithm enhanced with alpha-beta pruning. Our agent is consistent in achieving the best possible outcomes, and it will either beat opponents that play randomly or tie with opponents that play optimally. This shows us that the agent uses the optimal strategy and was implemented correctly. The alpha-beta pruning improved the efficiency of the agent without sacrificing the quality of its decisions. Even though Tic Tac Toe is a simple game, the techniques we used (adversarial search, game tree evaluation, pruning) are fundamental and can be implemented in more complex games and AI systems. Overall, this project demonstrates the power of systematic search strategies in building intelligent agents that are capable of making optimal decisions.

7 Future Work

In the future, we can implement a depth-limited search if we expand this project beyond Tic Tac Toe to a more complex game such as Othello (Reversi). While our current minimax algorithm with alpha-beta pruning performs well for small games with limited game trees, it becomes inefficient in games with significantly more possible states. Othello has a much larger branching factor and deeper game trees, making a full minimax search impractical without further optimization. In this context, alpha-beta pruning becomes substantially more valuable—while in Tic Tac Toe it may only offer marginal speedups, in Othello it can significantly reduce the number of nodes explored.

To adapt our current minimax implementation we would need to add a depth limitation. Instead of running minimax until terminal game states such as win/loss/draw, we would need to restrict the recursion to a fixed depth. Once this depth is reached, we would evaluate the board using a heuristic function that approximates how favorable the position is for the player. We would also need to design a heuristic function suitable to Othello. Some possible evaluation factors include the number of discs controlled by each player, control of corner positions (which are very strong in Othello), and the number of legal moves available to the player. By adapting our current minimax algorithm to a more complex game like Othello, we can make the algorithm more scalable and efficient.

References

Barr, A. and Feigenbaum, E. A. (1981), *The Handbook of Artificial Intelligence*, Vol. 1, William Kaufmann.

Knuth, D. E. and Moore, R. W. (1975), ‘An analysis of alpha-beta pruning’, *Artificial Intelligence* **6**(4), 293–326.

Slagle, J. R. and Dixon, J. K. (1969), ‘Experiments with some programs that search game trees’, *Journal of the ACM* **16**(2), 189–207.

von Neumann, J. (1928), ‘Zur theorie der gesellschaftsspiele’, *Mathematische Annalen* **100**(1), 295–320.