CSCI4511W Final Project

Jun Yeol Ryoo
ryoo0005@umn.edu
University of Minnesota
Minneapolis, Minnesota, USA

1 Abstract

Various approaches can be used for solving a given problem in everyday problems. Numerous algorithms can be considered to be used in solving a problem in particular within the realm of computer science. Therefore, that is the reason why considering diverse approaches in solving a problem is crucial as certain approaches might be suitable for some specific problems while unsuitable for others. In this paper, various methods for developing an artificial intelligence (AI) agent for complex games are explored, with a specific focus on the Monte Carlo Tree Search and the Minimax algorithm with Alpha-Beta Pruning. Moreover, this paper will specifically concentrate on the 3D tic-tac-toe game as one of the complex games. Additionally, there will be a review of the background concerning the adversarial search problem, which is directly relevant to creating an AI agent for the 3D tic-tac-toe game. Furthermore, the paper not only delves deeply into analyzing each method, namely the Monte Carlo Tree Search and the Minimax with Alpha-Beta Pruning, used in developing an AI agent for the game but also offers a comprehensive comparison among them.

2 Problem Overview

The 3D tic-tac-toe game, also known by the trade name Qubic, is a two-player board game, follows a turn-based game rule similar to various other board games. In contrast to the traditional tic-tac-toe, which features a 3x3 grid totaling nine cells, the 3D tic-tac-toe comprises a total of 64 cells organized across four separate boards, each with a 4x4 cell layout. Figure 1 shows the image of the board of the 3D tic-tac-toe game. Similar to the traditional version of the game, a winning row can be established horizontally, vertically, or diagonally within a single board. However, as there are 4 boards in total, the 3D tic-tac-toe introduces additional winning possibilities, including vertical columns or diagonal lines that span through multiple boards simultaneously. Even though a 3x3x3 version of the game exists, the 4x4x4 version is more widely embraced due to its subtle differences in gameplay. The 3x3x3 version lacks the possibility of ending in a draw, and unless a rule is imposed to prevent the first player from taking the center cell, it often results in a higher likelihood of victory for the player who starts the game first. However, this rule also introduces an imbalance, making it easier for the second player to win the game. Those factors are the reason why the 4x4x4 version is generally preferred. In contrast to the traditional tic-tac-toe, which has only 8 winning lines, the 3D version with 4x4x4 cells introduces more complicated game states with numerous winning possibilities. To be specific, each board has the same 8 winning lines as the traditional version. With a total of 4 boards, this accounts for 32 (8x4) winning lines. In addition, the 3D tic-tac-toe introduces vertical lines, where each of the 16 cells has a corresponding line ascending from the bottom board through corresponding cells on other boards. Furthermore, each corner cell on the bottom board contributes to 3 additional winning lines through secondary lines originating from these positions. Additionally, each inner cell adjacent to those corner cells also forms its winning line originating from its position. Consequently, the 4x4x4 tic-tac-toe has a total of 76 winning lines. As evident, the 3D version presents a significantly more complex and strategically demanding game compared to the traditional version. As a result, the complexity of the 4x4x4 tic-tac-toe game underlines the significance of carefully selecting appropriate approaches when developing an artificial intelligence agent for the game. Unlike simpler games, the immense number of possible state spaces makes it exceptionally difficult to play. For instance, employing the mini-max algorithm, one of the feasible approaches for the traditional 3x3 tic-tac-toe with 9 cells, becomes impractical for the 3D version. This is because the immense size of the state space, accompanied by a significant branching factor, makes the approach inefficient. Therefore, the necessity for adopting appropriate methodologies becomes more evident especially when developing an AI agent for such a complex game. Therefore, this paper delves into the discussion of several algorithms that are for addressing the challenges posed by complex games characterized by extensive state spaces. Those algorithms include the minimax algorithm with an adeptly chosen evaluation function, the minimax algorithm enhanced by alpha-beta pruning, and the Monte Carlo Tree Search. Beyond providing an in-depth analysis of each of those algorithms, the paper further explores comparative evaluations to determine the most effective approach for the given complex game state space, 3D tic-tac-toe.

3 Background

To start with, the 3D tic-tac-toe game is one of the various adversarial and positional board games. An adversarial game, also known as a competitive game, is a type of game in which two or more players have conflicting goals, winning the game. In the game, both players try to maximize

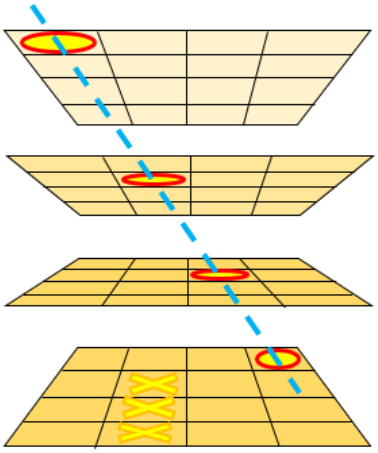

Figure 1. Example of traditional 3D Tic-Tac-Toe game boards with 4x4x4 cells. The blue dashed line represents winning line spanning through mutiple boards.

their winning conditions while minimizing their opponent's chances of winning. As there can be only one winner in the game, they need to make a well-structured strategy to win the game. In other words, each player needs to be careful in choosing their actions, considering both their own potential outcomes and their opponent's possible responses. The Tic-tac-toe game is a representative game of an adversarial game as players try to make a winning line in the game while trying to block opponents from making their winning line. Positional board game is a broad term used to describe games in which the primary focus is on controlling or occupying specific positions on the board. This characteristic of the game puts a difference between positional board games and other types of board games like movement-based games or resource-management games. To be specific, a positional board game has some characteristics which are static or grid-based board, strategic placement, and limited movement respectively. As a positional board game is played on a board with fixed positions or a grid format, it is regarded as static. Winning from the positional board game relies on strategically placing a piece in specific locations to control key areas, block opponents, or create advantageous lines that contribute to getting a win from a game. Even though pieces might have some movement abilities, it's usually restricted or not the primary focus of the game and that is the reason why it is called having limited movement. As a 3D tic-tac-toe game has those two characteristics, it is categorized as an adversarial and positional board game. Adversarial and positional board games are studied in various fields, including game theory, and also in artificial intelligence. Therefore, it is not hard to find related papers or studies concerning those games. For making an AI agent for the game, deciding which modeling approach to use can be regarded as the very first step to start on. Selecting a specific way of representing and understanding positional board games could affect the overall result of the outcome in making an AI agent for a game.

3.1 Modeling Approaches

There is an article studied on one of the modeling approaches not only for a specific board game but also for various positional board games [1]. In the article, they defined a board as a composite algebraic structure that consists of positions and rules. The introduced modeling approach from the article contains not only the position set itself but also a subset of the position set, which is called as win-groups set in this article. The elements of the win-groups set mean winning combinations from the positional board game. Therefore, a win-groups set can be interpreted as winning lines in a tic-tac-toe game. These components play a crucial role in defining the characteristics of a board, particularly in the context of game playing. Moreover, recent advancements in artificial intelligence and machine learning present intriguing possibilities for game rule interpretation. In a novel study, an interpretable AI, leveraging low-degree Zhegalkin polynomials, has emerged [2]. According to the paper, the method allows for the automatic learning of game rules, providing an exact representation in polynomial time. Significantly, this representation consumes less storage than a special class of Boolean functions, making it not only efficient but also interpretable, offering a promising way for future developments in the modeling approach.

3.2 Monte Carlo Tree Search

In addition to selecting suitable modeling approaches for board games, it is equally critical to choose appropriate methods tailored for positional board games characterized by significantly vast state spaces and high branching factors. This is due to the fact, as previously mentioned, that there are situations where discovering solutions is nearly impossible when relying on basic algorithms without any additional techniques, such as the min-max algorithm without enhancements. While the min-max algorithm can be effective in addressing problems with relatively small state spaces, adapting more advanced algorithms becomes necessary when dealing with complex games with vast state spaces. One potential method for addressing complex state spaces is the use of Monte Carlo Tree Search (MCTS). MCTS is one of the computational algorithms used in decision-making processes, particularly in game-playing scenarios, and is well-known for its application in software designed for board games like Go and Chess. For the brief explanation of the algorithm, it maintains a tree structure where each node represents a state in the game, and the edges represent possible moves or actions from a node. Starting from the root of the tree, the algorithm traverses down the tree using a selection strategy. The goal is to find the most promising node to explore further. Once a node is selected, it expands the

tree by adding child nodes corresponding to possible moves from the current state. Following the selection of child nodes, the algorithm simulates the game until it reaches the end of the game. The result of the simulated gameplay is then backpropagated through the selected nodes, updating statistics for each node in the path. Those steps are repeated until a termination condition is met. Upon meeting specific termination conditions, the algorithm finally makes a decision based on the statistics accumulated during the search. Even though the algorithm is well-suited for games with extensive state spaces, it is not universally applicable to all such games. The article suggests that modifications or integration with other additional techniques may be necessary when tackling more complex problems. To delve into those techniques, there is another article focused on MCTS with two distinct types of enhancements [3]. One of them is the Rapid Action Value Estimation (RAVE) algorithm, which utilizes an all-moves-at-first heuristic to estimate each action's value. According to the article, RAVE enables the sharing of knowledge among related nodes, leading to a faster estimate of action values. The second enhancement involves a heuristic function learned through self-play and temporal difference learning. This heuristic is then employed to initialize values for new positions in the search tree. In addition to these, there is another enhanced MCTS algorithm designed not only to reduce the time required for MCTS but also to enhance overall performance [4]. This algorithm is combined with the MTD(f) algorithm, a simpler and more efficient alternative to the regular minimax algorithm, to mitigate the impact of the randomness of Monte Carlo on search results. As can be seen from various articles, conventional MCTS and its enhanced version are widely used in various complex games.

3.3 Pruning

In conjunction with these advanced methods, pruning stands out as a widely employed and effective alternative approach in adversarial and positional board games. The pruning technique identifies and eliminates unproductive branches in the game tree, which are those not likely to lead to a winning outcome. This selective elimination significantly reduces the computational burden associated with exploring all possible moves, making pruning particularly valuable in games with vast state spaces and complex decision trees. The key concept behind pruning is to intelligently trim away portions of the search space that are unlikely to result in a favorable outcome, thereby focusing computational efforts on more promising paths. While Monte Carlo Tree Search and its enhancements offer powerful strategies, pruning also provides a distinct way for optimizing decision-making processes in complex games. Just as with MCTS, there exist various pruning strategies including Alpha-Beta pruning, Enhanced Transposition Cutoff, and other heuristic-based approaches. Alpha-Beta pruning, for instance, exploits the fact that some branches of the tree can be disregarded without evaluating every possible move, significantly reducing the number of nodes explored. Like MCTS, the application of pruning techniques in adversarial and positional board games is extensive. Numerous articles explore various aspects of pruning techniques. One such article introduces a new pruning algorithm for improving the efficiency of pruning and position evaluation [5]. To be specific, the new pruning algorithm comprises two distinct types of pruning techniques: one focuses on nodes classified as win or lose, while the other targets sibling nodes of the first kind. The assessment of positions involves both the evaluation function and the position classifier. The experimental results demonstrate the enhanced efficiency of the new algorithms in terms of tree pruning and position evaluation. The conventional belief regarding the search tree suggests that better decisions can be achieved by going deeper into the search. However, an intriguing paper challenges this conventional belief [8]. According to this research, lookahead pathology contradicts the belief that deeper searches consistently result in better decisions, as there are instances where deeper searches lead to worse decisions. Furthermore, this paper argues that every game exhibits local pathologies, indicating specific sections where deeper searches result in a decrease in decision quality. To address this issue, a modified search algorithm called error-minimizing minimax (EMM) was introduced. EMM is designed to identify and mitigate the impact of pathological sections in the search at a shallower depth. The experimental results demonstrated that EMM outperformed the traditional minimax algorithm, exhibiting fewer pathological characteristics. Despite the maturity and success of the Alpha-Beta algorithm in game-playing programs, a paper introduces several enhancements for further improvement [6]. One of the proposed enhancements is the Enhanced Transposition Cutoff (ETC). This improvement aims to maximize the benefits derived from the transposition table by performing additional lookups for all successors of a node. Once the lookup produces a value sufficient for cutoff, the search can be halted at that node. The paper suggests that implementing ETC in a chess game can lead to a significant reduction of the search tree, estimated to be around 28%. There is another paper introducing an enhancement to the Alpha-Beta pruning algorithm [7]. To be specific, the focus of the paper is on the order in which child nodes are considered during the expansion of the game tree. This enhanced algorithm prioritizes nodes with larger estimated values, leading to more efficient pruning. The results demonstrate that, as the depth of the tree increases, the enhanced Alpha-Beta search prunes a greater number of nodes, resulting in substantial time savings during the searching process.

4 Approach

As discussed earlier, numerous articles explore diverse approaches for adversarial and positional baord games. When developing an artificial intelligence agent for playing 3D tic-tac-toe on a 4x4x4 grid, the adoption of advanced algorithms becomes inevitable due to the complexity of state spaces. In this paper, the author adopted the Monte Carlo Tree Search algorithm and Minimax with Alpha-Beta pruning, utilizing two distinct evaluation functions. The focus extends beyond merely creating an AI agent for the game; it involves a comparative analysis of outputs generated by different algorithms. Additionally, the paper delves into the examination of two different evaluation functions in conjunction with Alpha-Beta pruning.

4.1 Programming Language

The decision to implement the AI agent and algorithms in C++ was driven by a thoughtful consideration of the programming language's unique advantages. While Python is widely acknowledged for its extensive AI libraries, which offer diverse functionalities, user-friendly APIs, and well-documented tutorials for both beginners and experienced developers, C++ was chosen for its inherent speed benefits. C++ is a compiled language, in contrast to Python's interpreted nature, resulting in faster execution times and improved performance. Given the need for multiple program executions to ensure a precise and thorough comparison between the earlier-mentioned algorithms, the author determined that the speed advantage offered by C++ would be beneficial for this specific project.

4.2 Board Representation

To start with, a crucial step in implementing the 3D tic-tac-toe game involves creating the game board. This foundational element is essential because subsequent algorithms, including Monte Carlo Tree Search and Alpha Beta Pruning, rely on the board to determine the current state of the game. Consequently, the initial decision revolves around selecting the approach for representing the game board. Various data structures are available for implementing the game board. For instance, an array can serve as a representation, and it's not confined to the use of multidimensional arrays; a single array can also be adopted for representing the game's board. In other words, the choice of representation indeed lies in the developer's preference. Apart from arrays, alternative data structures like nested dictionaries and custom objects are also possibilities. In this project, a multidimensional vector of characters was chosen to represent the game board. This is because, in C++, a vector is renowned for its flexibility and simplicity in data management.

4.3 Approach for AI agent

As mentioned earlier, two algorithms are employed to create an AI agent for playing 3D tic-tac-toe: Monte Carlo Tree Search and Minimax with Alpha Beta Pruning.

4.3.1 Monte Carlo Tree Search. To start with, when adopting the Monte Carlo Tree Search for an AI agent, it is crucial to set an appropriate maximum time for the algorithm to run the simulation. This directly influences the accuracy of the desired results, as the algorithm conducts simulations within a specified time frame. However, setting an excessively high designated maximum time may result in suboptimal game performance. This is because the AI agent will consume a significant portion of the overall gameplay when the maximum time is set too high. Although this situation can yield accurate results due to numerous simulations, it has a negative impact on the overall game environment, leading to high occupancy of game time. Therefore, the balance between achieving high accuracy and rapid decision-making needs to be carefully considered when employing Monte Carlo Tree Search. For this project, the maximum time for the algorithm to run the simulation will be set to 20 seconds. Another factor directly impacting the algorithm's outcome is the method by which it expands child nodes. In this project, the expansion process involves performing proper initializations for every new node and subsequently randomly selecting which one to simulate.

4.3.2 Minimax with Alpha Beta Pruning. In a similar manner to the Monte Carlo Tree Search, the Minimax algorithm with Alpha-Beta Pruning is influenced by various factors, and one of them is the maximum depth parameter during the search process. Essentially, the maximum depth determines how many levels deep the algorithm delves into the game tree. A higher maximum depth empowers the algorithm to contemplate a greater number of potential future moves, resulting in a more profound and comprehensive exploration. However, increasing the maximum depth also increases the number of nodes that need to be evaluated. It can also be interpreted that the execution time can grow significantly as the branching factor of the 3D tic-tac-toe game tree is high. Consequently, for a similar reason, this project will set the maximum depth for the Minimax algorithm to explore at 7. Since the minimax algorithm is not feasible for exploring each node until reaching the leaf node, the evaluation function plays a pivotal role in treating non-leaf nodes as endpoints of the game. The minimax algorithm initiates pruning from the provided node, underscoring the significance of the evaluation function in Alpha-Beta pruning. In this project, a two-fold strategy is employed, leveraging two distinct evaluation functions to enhance the decision-making of the AI agent.

In the initial evaluation function, for each one of the 76 possible winning lines from the 3D tic-tac-toe game, it begins

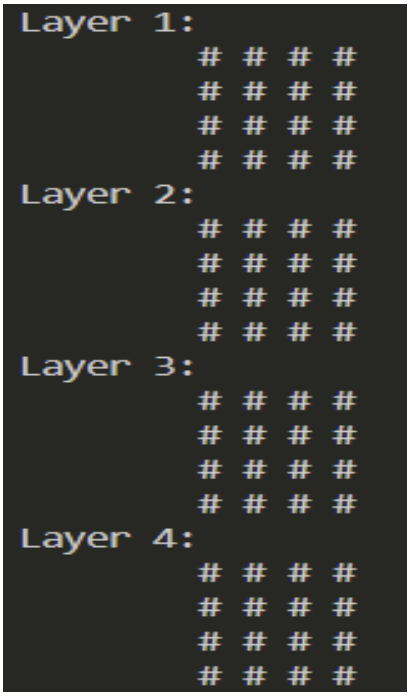

Figure 2. Representation of 3D tic-tac-toe board. '#' represents empty space from the board.

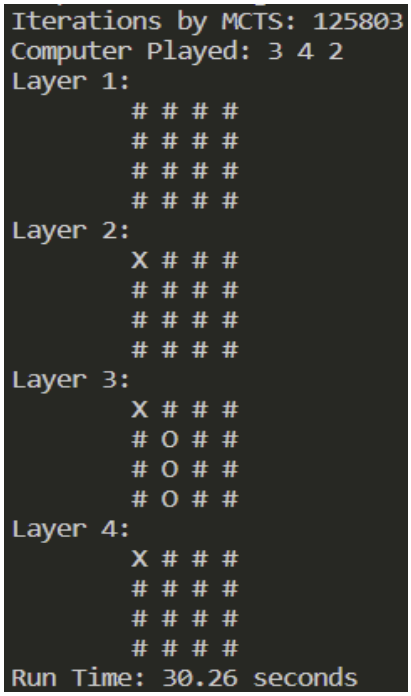

Figure 3. The result of using Monte Carlo Tree Search for AI agent with the maximum simulation time of 30 seconds.

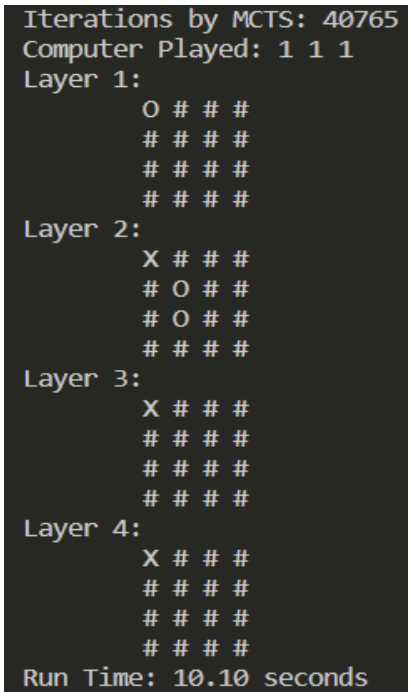

Figure 4. The result of using Monte Carlo Tree Search for AI agent with the maximum simulation time of 10 seconds.

to count how many X's and O's can be found. If there is 0 or 1 occurrence in a winning line, the score is simply set to 0. If there are 2 occurrences in a winning line, the score is set to 2. The score gets bigger as the number of occurrences gets higher. After adding all those scores for X and O separately, the evaluation function returns (X_score - O_score) as X is the maximizing player.

The second evaluation function is a slightly modified version of the first one. Specifically, if a single line contains 2 or 3 pieces of one player and even a single piece of the other player, both players receive a score of 0 because a tic-tac-toe cannot occur in such scenarios. For a simple example, let's imagine there are 2 X's and 0 O's on any of the 76 winning lines; in this case, 2 scores are added to X_score. However, if there is at least 1 opponent's piece on the line, both players receive a score of 0 from that specific line. The second evaluation function can be regarded as be more accurate option.

5 Experiment Design and Results

The project involved a variety of experiments. These experiments encompassed not only a comparison between two distinct algorithms utilized for creating an AI agent—Monte Carlo Tree Search and Minimax algorithm with Alpha Beta Pruning—but also included additional experiments on the same algorithm with subtle variations. To begin, Figure 2 illustrates the game board representation. Here, '#' denotes an empty space, while 'O' and 'X' represent pieces for player

Figure 5. The results depict 10 games played between two AI agents utilizing the Monte Carlo Tree Search algorithm. MCTS1 operated with a 15-second time limit, whereas MCTS2 had a 10-second constraint.

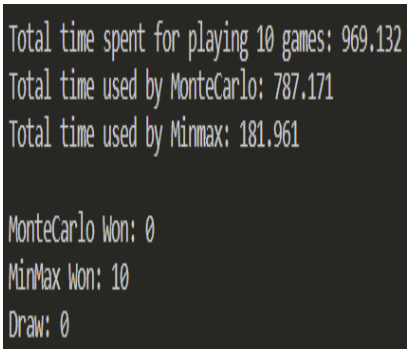

Figure 6. The comparison between AI agent employing Monte Carlo Tree Search and another AI agent utilizing Minimax with Alpha Beta Pruning in 10 game plays.

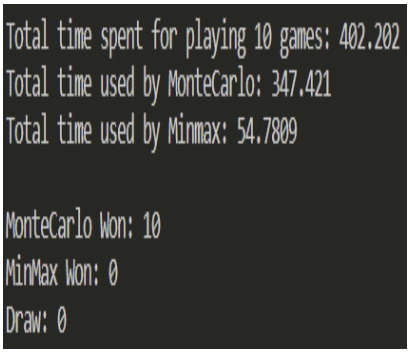

Figure 7. In contrast to Figure 6, the AI agent in this figure employed the first version of the evaluation function.

O and player X, respectively. To achieve victory in the game, a player must either create a winning line spanning across all four layers or establish a winning line within a single layer. To begin the analysis of the results obtained from employing Monte Carlo Tree Search in developing the AI agent, refer to Figure 3 and Figure 4. Figure 3 displays the outcomes under a maximum simulation time of 30 seconds, showcasing significantly higher iterations compared to the scenario presented in Figure 4. Both figures show the situation in which an X player can secure victory in the game by placing one more piece at (1,1,1). In other words, this scenario implies that the AI agent should strategically position an O piece to prevent X player from securing victory in the game. However, in Figure 3, the AI agent fails to secure this position, unlike the successful prevention achieved by the AI agent in Figure 4. These situations highlight a drawback in Monte Carlo Tree Search, specifically its susceptibility to randomness in the expansion process. Despite having more time for simulation in Figure 3, the inherent randomness results in an unexpected outcome. In this scenario, it becomes evident that depending solely on randomness in the expansion process of Monte Carlo Tree Search is insufficient to reliably achieve the desired result. To address this limitation, it becomes imperative to adopt more advanced methods to ensure the realization of the desired outcome. Another approach is to extend the time limitation for executing simulations. Despite the algorithm's dependence on randomness in the expansion process, increasing the time limitation allows for a greater number of simulations, thereby enhancing the likelihood of achieving the desired outcome through the accumulation of simulated results. Figure 5 validates this observation. To be specific, the figure presents the outcomes of 10 games played between two AI agents, each employing the Monte Carlo Tree Search (MCTS) algorithm—referred to as MCTS1 and MCTS2. In this experimental setup, MCTS1 was configured with a maximum simulation time limit of 15 seconds, while MCTS2 had a 10-second time limit. As depicted in the figure, MCTS1 conducted a significantly higher number of iterative simulations compared to MCTS2. Consequently, due to its increased simulation iterations, MCTS1 secured more victories than MCTS2. Nonetheless, as previously highlighted, the element of randomness occasionally allows MCTS2 to prevail over MCTS1 in certain games, despite conducting fewer iterations.

When comparing the performance of different algorithms, specifically Monte Carlo Tree Search and Minimax with Alpha Beta Pruning, please refer to Figure 6. The figure illustrates the results of 10 games played between an AI agent employing Monte Carlo Tree Search and another AI agent utilizing Minimax with Alpha Beta Pruning. For the AI agent with the Minimax algorithm, the second evaluation function was utilized for Alpha Beta Pruning. Despite the significantly higher time spent by Monte Carlo Tree Search, it fails to secure a victory in any of the games. To underscore the significance of the evaluation function in the Minimax algorithm, consult Figure 7. The only differing factor between Figure 6 and Figure 7 is the alteration of the employed evaluation function, transitioning from the second version to the first version. This alteration significantly influences the outcomes of 10 games played between them, as depicted in Figure 7,

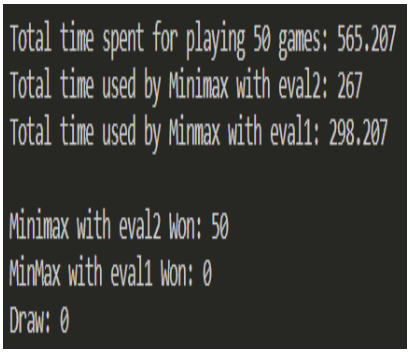

Figure 8. The figure presents the outcomes of games between two AI agents utilizing the Minimax algorithm with Alpha Beta Pruning. This illustration highlights the pivotal role of the evaluation function within the Alpha Beta Pruning.

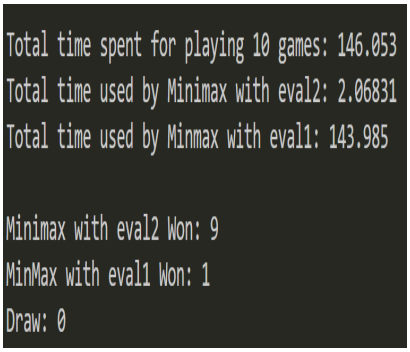

Figure 9. The figure depicts the outcomes of 10 games played between two AI agents utilizing the Minimax algorithm with Alpha Beta Pruning, each with distinct maximum depth settings for their Alpha Beta Pruning searches.

where the AI agent with the Minimax algorithm could not secure a victory in any of the games. To further emphasize the critical role of a well-designed evaluation function in the Minimax algorithm with Alpha Beta Pruning, refer to Figure 8. This figure displays the outcomes of 50 games played between two AI agents: one employing the first version of the evaluation function and the other utilizing the second version. Both agents operated with a maximum depth of 7. Despite both agents allocating a comparable amount of time during their respective turns, there is a noticeable difference in the game results. Specifically, the agent using the second evaluation function with Minimax won all 50 games, without a single loss. Figure 9 presents a distinct scenario. Specifically, 10 games were played between two AI agents employing the Minimax algorithm with Alpha Beta Pruning. The AI agent utilizing the first evaluation function had a maximum search depth set to 7 in Alpha Beta Pruning, while the other agent had its depth restricted to 3. Notably, Figure 9 illustrates that despite the considerably shorter time spent by the AI agent with the second evaluation function, it emerged victorious in the majority of the 10 games played. Those figures illustrate that the ability to select a high-quality evaluation function for the Minimax algorithm with Alpha Beta Pruning is directly linked to achieving the desired results.

6 Analysis of the results

To determine the effectiveness of AI algorithms in game-playing scenarios, the project performed a thorough investigation of them, with a special emphasis on Minimax with Alpha Beta Pruning and Monte Carlo Tree Search. Figure 2 served as the foundational reference, offering a visual representation of the game board. Symbol '#' denoted vacant spaces, while 'O' and 'X' represented player moves. As the analysis progressed, there was a focus on comprehending the AI agent utilizing the Monte Carlo Tree Search algorithm. To start with, figures 3 and 4 offer valuable insights into its operation. Figure 3, characterized by a 30-second simulation time, revealed the inherent challenges posed by the algorithm's reliance on randomness during expansion processes. Despite an extended simulation duration compared to Figure 4, it resulted in the undesired result from the game scenario. However, Figure 4 presented a contrasting scenario, showcasing a moment where the algorithm's strategic capabilities successfully prevented the opponent from achieving victory. Despite the inherent challenge of depending on randomness during expansion processes, subsequent analysis indicates that this challenge can be mitigated to some extent by extending the maximum time limit for simulations. Furthering the exploration, a comparative analysis was conducted to contrast Monte Carlo Tree Search with Minimax using Alpha Beta Pruning, as illustrated in Figure 6. This comparison revealed an intriguing outcome. Even though the AI agent using Monte Carlo Tree Search used a notably longer time than the one using Minimax, it failed to secure a single victory against the AI agent utilizing Minimax. The subsequent experiment presented a contrasting scenario, as depicted in Figure 7. When the evaluation function for the AI agent employing Minimax shifted from the second to the first evaluation function, the Monte Carlo Tree Search-based AI agent secured victories in all 10 games. This experiment underscores the critical dependence of the Minimax algorithm with Alpha-Beta Pruning on the quality of its evaluation function. To further validate it, an additional experiment was conducted. In the experiment, two AI agents utilizing the Minimax were involved in playing games. One agent employed the first evaluation function, while the other utilized the second version of it. After a total of 50 games between them, the AI agent using the second evaluation function secured victories in every single game without a single loss. Furthermore, to reinforce the results of the experiment, an additional 10 games were played between the two AI agents. This time, the AI agent utilizing the second

evaluation function was restricted to searching the game tree up to a maximum depth of 3. Conversely, the other AI agent, equipped with the first evaluation function, could explore the game tree up to a depth of 7. As depicted in Figure 9, despite the majority of the gameplay time being consumed by the AI agent using the first evaluation function, it managed to secure a victory in only one out of ten games. In summary, these experiments underscore the critical importance of selecting an appropriate and precise evaluation function for Alpha-Beta Pruning within the Minimax algorithm to achieve the desired outcomes.

7 Conclusion

In this paper, the exploration began with an introduction to the challenge of employing an AI agent for playing 3D tic-tac-toe, offering a comprehensive understanding of the game, particularly for readers unfamiliar with its 4x4x4 grid structure. Subsequently, the paper delved into a review of relevant literature, focusing on methodologies applicable to creating AI agents for adversarial and positional board games. This review primarily focused on two techniques: the Monte Carlo Tree Search and various pruning methods. With a solid foundation established, attention shifted towards practical applications. The core of the investigation centered on making an AI agent specific to the 3D tic-tac-toe game. This paper also conducted diverse experiments and analyses. The experiments clarified the key aspects of AI performance. The paper first examined how AI agents using the Monte Carlo Tree Search method performed. This highlighted challenges when relying too heavily on the inherent randomness of this approach during its expansion phase. Additionally, the paper broadened its focus to compare outcomes from AI agents using Monte Carlo Tree Search and the Minimax algorithm with Alpha-Beta Pruning. These tests incorporated specific adjustments, like varying the depth of tree searches and using different evaluation functions during the pruning stages. Overall, the paper emphasizes the importance of selecting and designing an accurate evaluation function for the pruning process. After conducting various experiments, the analysis indicates that the selection of the algorithm for developing an AI agent for the 3D tic-tac-toe game can vary. Specifically, when an evaluation function accurately assesses a particular state, selecting a Minimax algorithm with Alpha-Beta Pruning may be more advantageous than using Monte Carlo Tree Search. This observation comes from the analysis, which indicates that the Minimax algorithm, when paired with effective evaluation functions, achieved desired results efficiently and rapidly compared to Monte Carlo Tree Search. Conversely, in the absence of such an evaluation function, utilizing Monte Carlo Tree Search with a suitable maximum time limit for the simulation process could serve as a viable alternative.

References

[1] Sergio Antoy. 1987. Modeling and Isomorphisms of Positional Board Games. *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-9, 5 (1987), 669–675. https://doi.org/10.1109/TPAMI.1987.4767961

[2] Jared L. Aurentz, Adrián Martínez Navarro, and David Ríos Insua. 2022. Learning the Rules of the Game: An Interpretable AI for Learning How to Play. *IEEE Transactions on Games* 14, 2 (2022), 253–261. https://doi.org/10.1109/TG.2021.3066245

[3] Sylvain Gelly and David Silver. 2011. Monte-Carlo tree search and rapid action value estimation in computer Go. *Artificial Intelligence* 175, 11 (2011), 1856–1875. https://doi.org/10.1016/j.artint.2011.03.007

[4] Zhongzhi Li, Hedan Liu, Yuechao Wang, Jiankai Zuo, and Zeyuan Liu. 2020. Application of Monte Carlo Tree Optimization Algorithm on Hex Chess. In *2020 Chinese Control And Decision Conference (CCDC)*. 3538–3542. https://doi.org/10.1109/CCDC49329.2020.9164656

[5] Hal-Tao Liu and Bao-En Guo. 2012. A new pruning algorithm for game tree in Chinese Chess Computer Game. In *2012 International Conference on Machine Learning and Cybernetics*, Vol. 2. 538–542. https://doi.org/10.1109/ICMLC.2012.6358980

[6] Jonathan Schaeffer and Aske Plaat. 1999. New Advances in Alpha-Beta Searching. (10 1999). https://doi.org/10.1145/228329.228344

[7] Zhang-Congpin and Cui-Jinling. 2009. Improved Alpha-Beta Pruning of Heuristic Search in Game-Playing Tree. In *2009 WRI World Congress on Computer Science and Information Engineering*, Vol. 2. 672–674. https://doi.org/10.1109/CSIE.2009.527

[8] Inon Zuckerman, Brandon Wilson, and Dana S. Nau. 2018. Avoiding game-tree pathology in 2-player adversarial search. *Computational Intelligence* 34, 2 (2018), 542–561. https://doi.org/10.1111/coin.12162 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/coin.12162