American University of Armenia

Zaven and Sonia Akian College of Science and Engineering

Capstone Project

Quarto Game Development

Student: Zara Karapetyan
Supervisor: Monika Stepanyan

Spring, 2025

Abstract

This project focuses on the implementation and analysis of artificial intelligence strategies for the Quarto board game. Quarto is a two-player strategy game where the goal is to align four pieces that share a common attribute. A unique twist of the game is that each player selects the piece their opponent must place, which makes decision-making especially important.

The project was first developed in Java and then integrated into a Unity environment using C#, where the working game was designed and presented with a visual interface. Multiple performance tests were conducted across different stages of gameplay to analyze how well the algorithms performed in terms of speed and decision-making.

Contents

Chapter 1
Introduction

Solving various board games by developing different AI-based strategies is considered a valuable research area in game theory and artificial intelligence because it helps test complexities and find faster solutions. Two-player games provide a particularly effective framework for analyzing game theory principles and AI strategies. Among two-player, very well-known games, there is also one game named Quarto, which, though not as well known as Chess or Tic Tac Toe, can still serve as a perfect instrument for developing and testing different AI strategy approaches and decision-making processes due to its unique and diverse characteristics.

As I mentioned, Quarto differs from others by its unique characteristics and rules. First, it is a perfect information and zero-sum game, which means that both players can see everything happening in the game, and one player's gain is the other player's loss, so there's no situation where both players benefit [2]. The goal in Quarto is to create a line(any row, column, or diagonal) where all the pieces will have at least one common feature. The game is played on a 4x4 grid, with 16 spaces for each unique piece. The four characteristics of each piece are tall/short, light/dark, round/square, and solid/hollow. There are a total of 16 possible unique combinations [1]. The unique twist in the game is that on each turn, your opponent chooses the piece you have to place. So, when the game starts, the first player selects the piece and gives it to the second player to put on the board. Then, the second player chooses a piece and hands it to the first player to place it, and so on.

[Figure 1]: Quarto Board

Despite being a valuable tool for experimenting with various AI algorithms, Quarto hasn't been thoroughly examined yet. There are not many academic articles or publications connected with it. Most research focuses primarily on search algorithms, such as minimax and alpha-beta pruning, without fully considering their effective backtracking strategies or the development of an appropriate evaluation function. This challenged me since I want to create efficient backtracking search algorithms and a suitable evaluation function that will allow for a fairer comparison of different AI systems while being as effective as possible. My research tries to address the gap by analyzing and improving Quarto's AI decision-making processes.

Chapter 2
Literature Review

The research on the topic identifies several papers and publications that were useful in understanding Quarto's programming and mathematics techniques. Although there weren't many articles, they did assist me in better understanding how the game engine can be organized. They helped me improve my critical thinking skills in strategic gameplay and evaluation function design. Some publications focused on AI and decision-making processes, while others took a more mathematical approach. Collectively, they provided some direction for my implementation.

2.1 Insights from Russell & Norvig – Chapter 6: *Adversarial Search and Games*

In the initial stage of literature discovery, Artificial Intelligence: A Modern Approach by Stuart J. Russell and Peter Norvig (4th ed., 2022) book, especially Chapter 6, titled *Adversarial Search and Games*, was consulted and resulted in a preliminary understanding of the necessary topics [2] .

It introduces adversarial search problems and highlights the importance of treating opponents as smart decision-makers by which it starts discovering different game-tree search strategies and their essential components like minimax search algorithm,alpha-beta pruning, the importance of move ordering, heuristic alpha-beta tree search, evaluation functions, cutting off search, forward pruning, and Monte Carlo tree search.

Though the book does not pay attention to Quarto separately, the general principles and techniques discussed helped me a lot in designing AI for the game, especially in developing the evaluation function and optimizing the search processes.

2.2 Developing Strategic and Mathematical Thinking via Game Play: Programming to Investigate a Risky Strategy for Quarto

Peter Rowlett's article gives a valuable and straightforward look at the Quarto [3]. The work directly examines Quarto as a mathematical and strategic exercise. Rowlett explains three strategies they have implemented in Python, each showing different behaviors during gameplay. These include random, naive, and risky strategies.

The random strategy selects both a piece to place and a piece to give completely at random, without any evaluation of the board.

The naive strategy checks whether the piece handed over can lead to a win and plays it if possible; otherwise, it selects a move at random. When selecting a piece to give to the opponent, the strategy avoids handing over one that would allow an immediate win. Otherwise, it chooses the piece randomly.

The risky strategy builds upon the naive approach by introducing more strategic play. If the piece received from the opponent offers a winning move, the player takes the winning move. If not, the player tries to place it in a way that contributes to a line of pieces sharing at least one attribute, like those sharing a color or shape, to improve the chances of winning later. If no such placement is possible, the player makes a random move. When selecting a piece to give to the opponent, the strategy avoids giving a winning piece; otherwise, it chooses randomly. While this strategy can be effective, it is risky because trying to build such lines may also benefit the opponent, potentially creating winning opportunities for both players [3].

While testing each strategy multiple times, it turned out that the risky strategy performed better than the naive strategy.

2.3 A Flexible Extended Quarto Implementation Based on Combinatorial Analysis

A unique implementation of the extended Quarto game was developed by Daniel Castro Silva and Vasco Vinhas in 2008, with a strong emphasis on its extensibility, modularity, and educational possibilities [4].

The authors designed the system using a client-server architecture, where the game logic was implemented in Prolog (in particular, using the SICStus development environment), and the graphical user interface was developed in C++ with OpenGL [4]. The communication between the logic and the interface was handled using a simple TCP/IP-based socket protocol.

The implementation has a feature to save, replay, and resume games, which gives a better opportunity for strategic analysis, research, and education. Additionally, the developers included L- and T-shaped win conditions outside traditional rows, columns, and diagonals, increasing the whole count of winning combinations from 19 (some variations permit 2x2 squares as well) to 91, which made the game even more difficult and enjoyable. These additions gave a better framework for examining AI decision-making strategies and more chances for experienced players to explore various deeper strategies.

To improve gameplay, the authors introduced a heuristic called Quarticity. Rather than checking all possible outcomes, it estimates how many ways a player could form a winning situation based on the current board and the pieces left. They improved the evaluation by giving more importance to situations where a player is close to winning. For example, when only one piece is missing to complete the winning line. In contrast, early-stage setups with only a few matching pieces were considered less important [4].

In the paper, the authors explained how they used Quarticity to build five AI levels. The first played randomly, the next two either minimized or maximized Quarticity, the fourth used a basic minimax strategy, and the last used a deeper minimax with alpha-beta pruning. They then presented the findings of their analysis, where I noticed that human players who played against the computer (using the AI levels that had been developed) were able to learn the game rules more quickly than those who used traditional methods like playing on a physical board against other humans [4].

2.4 AI for the Board Game Quarto in Java

Another interesting project with a unique approach was presented in 2013 by Jochen Mohrmann, Michael Neumann, and David Suendermann. They developed an artificial intelligence for the game Quarto in Java [5].

They used a depth-first search for the decision-making process and the NegaMax Alpha-Beta pruning algorithm to improve runtime performance. NegaMax is similar to the Minimax algorithm, but instead of having separate maximization and minimization steps, it uses one recursive function that always tries to maximize the score but flips the score when it's the opponent's turn.

Additionally, they implemented a heuristic evaluation function that counted the number of lines with three pieces sharing at least one property, as these lines had the best chance of winning. The more of these lines a given piece could complete, the higher its heuristic value. The evaluation also considered whose turn it was, helping the AI better judge whether a position was favorable or dangerous depending on the player.

To reduce computational complexity, the authors mentioned in the article that they took advantage of certain properties of the game Quarto [5]. For example, they recognized that many board positions could be reached through different move sequences and used transposition tables to store and reuse previously computed evaluations, or they used the game's symmetrical nature to reduce the number of unique board states that needed to be evaluated.

In the end, the authors conducted performance analyses. They concluded their program could win the majority of human players, even experienced ones, winning 34 out of 50 games during testing.

2.5 Conclusion

Exploring these sources and seeing how they were built gave me a better grasp of designing my own evaluation function for Quarto. It made the concept of heuristics feel much more clear. Especially the Quarticity heuristic gave some insights into how to estimate the possibility of winning situations on the board and gave ideas for implementing my evaluation function. Furthermore, the method of gradually developing and testing the system from random players to more complex algorithms was a good approach that I also used in my implementation.

Additionally, I adopted transposition tables in my AI logic after reviewing Mohrmann et al.'s Java-based Quarto AI. I found this technique particularly effective in improving search efficiency by reusing evaluations of already evaluated moves.

Chapter 3
Methods

This chapter discusses the architecture of my implementation. Section 3.1 introduces the technologies used, Section 3.2 presents the hierarchical structure of the modules, and Sections 3.3 and 3.4 describe the modules' internal components and functionality.

3.1 Chosen Technologies

I used the Java programming language to implement the core logic of *Quarto*, including experimentation with search algorithms and evaluation functions. As Java provides a strong and flexible foundation for object-oriented programming, it made the development of the game's decision-making logic more effective and organized.

Additionally, for designing and implementing the game, I used Unity with C#, which allowed me to develop the game's environment and user interactions, as well as integrate and test different types of players, including a human player, random players, and AI players using the minimax and alpha-beta pruning algorithms.

3.2 Modular Structure

Below, I introduce the modular structure and components of my solution. Abstract classes and interfaces are **underlined**, and inheritance is indicated in **brackets**.

board

- Board
- Piece

player

- Player (*Abstract Class*)
- AIPlayer (Player)
    - RandomPlayer (AIPlayer)
    - MinimaxPlayer (AIPlayer)
    - AlphaBetaPlayer (AIPlayer)

search

- Agent (*Abstract Class*)
- MinimaxAgent (Agent)
- AlphaBetaAgent (Agent)
- Action

evaluation

- Evaluator (*Interface*)
    - SimpleEvaluator (Evaluator)

game

- GameEngine

util

- TranspositionTable

main

- Main

3.3 Core Game Formation: Board and Pieces

The **board** module contains two core classes (Board and Piece) essential for Quarto. They play a fundamental role in the game's structure, ensuring its rules and flow are appropriately maintained.

The **Board** class manages the overall game state in Quarto. It represents the 4x4 grid and maintains the game's current state, including the placed pieces, the current piece to be played, and the list of available pieces. It allows placing pieces on the board by ensuring the move is valid and updating the grid and the list of available pieces. The class also provides a method for generating valid actions based on the current piece; if a piece is selected, it generates "place piece on board" actions; if no piece is currently selected, it generates "give a piece to the opponent" actions. Additionally, the Board class provided functionality to check for win or draw conditions by evaluating rows, columns, and diagonals to determine whether four pieces in a line share a common attribute. Each cell in the grid holds a Piece object or remains empty if unoccupied. The board state can also be saved for making experiments or analyzing the behavior of search algorithms and evaluation functions from various game states.

The **Piece** class represents each of the 16 unique pieces using an 8-bit binary code where each bit stands for one of the four attributes, which are white or black, round or square, tall or short, and whether it has a hole or not. This way of representing pieces enables efficient attribute checks using bitwise operations. The class provides methods to check individual piece attributes and compare multiple pieces for shared attributes, essential for detecting winning conditions during gameplay.

3.4 Search Formation: Players, Agents, and Evaluation

The **player**, **search**, **evaluation**, and **util** modules are interconnected. They work together to help the AI make smart moves by analyzing future outcomes using search algorithms and heuristic evaluations instead of choosing randomly.

The core of the **player** module is an abstract class named **Player**, which serves as the foundation for all player types in the game. It defines two essential methods, *making a move* and *choosing a piece,* which every player must have in the Quarto game. The **AIPlayer** class extends **Player** and acts as a base for different types of smart strategies. It is further extended by classes **RandomPlayer**, which picks moves randomly, and **MinimaxPlayer** and **AlphaBetaPlayer**, which use appropriate search agents **(MinimaxAgent** and **AlphaBetaAgent)** to make more strategic decisions.

The **search** module provides the logic for strategic decision-making. It has an abstract class **Agent** that defines the structure of search-based algorithms. The Agent class is extended by two classes: **MinimaxAgent**, which uses the minimax search algorithm, and **AlphaBetaAgent**, which optimizes the minimax algorithm through alpha-beta pruning. Both agents rely on a helper class called **Action**, which represents a move (*placing a piece* or *giving a piece* to the opponent) and provides methods for *applying* and *reverting* moves, enabling efficient backtracking during search without the need to clone the board state.

In the **util** module, the **TranspositionTable** class maintains a transposition table that stores already evaluated board states and their scores. This helps avoid repeating the same calculations and makes both minimax and alpha-beta pruning faster.

The **evaluation** module provides the logic for scoring board states, helping the AI decide which moves are better. It includes an **Evaluator** interface, which defines a method for evaluating the board by assigning a score. The **SimpleEvaluator** class implements this interface, which uses basic rules to score the board. **MinimaxAgent** and **AlphaBetaAgent** use this evaluator to estimate the value of future game states during the search process.

The **GameEngine** class in the **game** module runs the Quarto game between two AI players. It handles turns, piece selection, move-making, and checks for a win or draw.

The **Main** class tests how different AI players perform against each other. It runs multiple games using the GameEngine, and counts win, losses, and draws and it prints the results after each set of matches.

3.5 Developing Quarto in Unity

After implementing the core logic of Quarto in Java, I integrated the project into Unity using C# to develop a fully playable and interactive game version.

Using Unity's game engine, I created the game board and visual representations of the pieces and implemented user interactions for gameplay. The game supports different player types, including a Human player, Random AI, and strategic AI based on Minimax and Alpha-Beta Pruning algorithms. The system handles player turns, win detection, and displays game progress in real-time.

The Unity version of the game is demonstrated through the following images.

Figure 3.1: Opening Scene

The game begins with an animated opening screen displaying the title "Quarto."

Figure 3.2: Main Menu Panel

The "Main Menu" panel lets players choose player types under "Options," view game instructions in "About," exit the game by pressing "Exit," or start playing by pressing the "Play" button.

Figure 3.3: Options Panel:Player Selection

The "Options" panel lets users select the player type for each participant. Dropdown menus provide choices such as Human, Random AI, and strategic AI based on Minimax or Alpha-Beta Pruning algorithms. The "Back" button allows users to return to the "Main Menu."

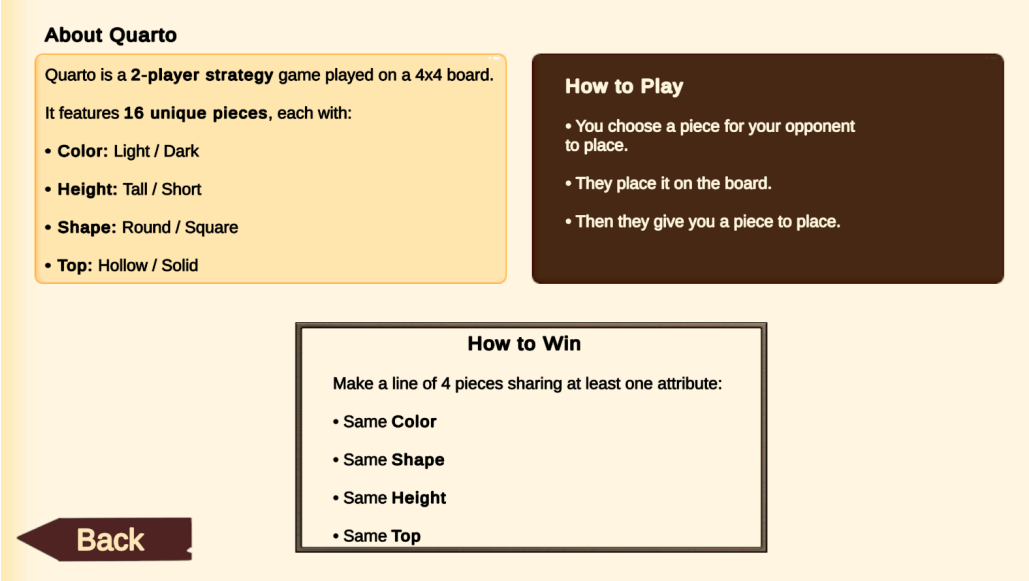

Figure 3.4: About Panel

The "About panel" contains three sections: a description of Quarto, instructions on playing, and the winning conditions. The "Back" button allows users to return to the "Main Menu."

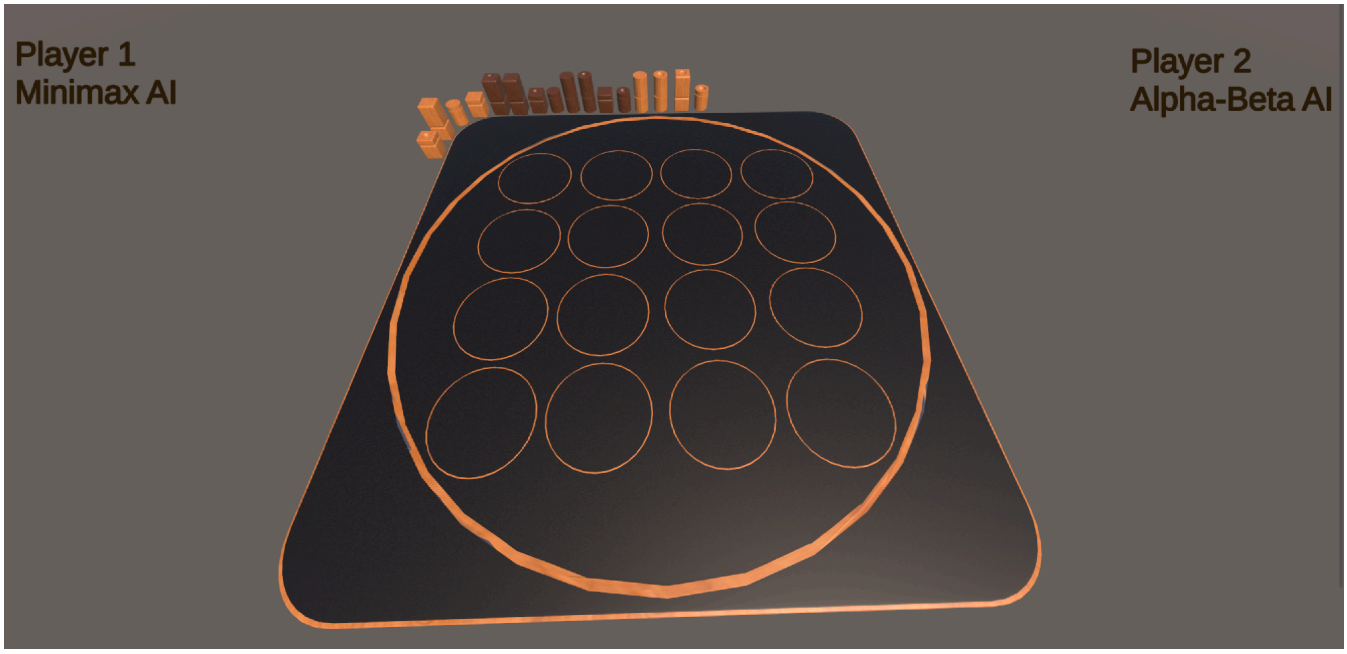

Figure 3.5: Game Start:Initial State

The game starts with an empty board. All 16 unique pieces are displayed and available for selection. The player who goes first must choose a piece for their opponent to place.

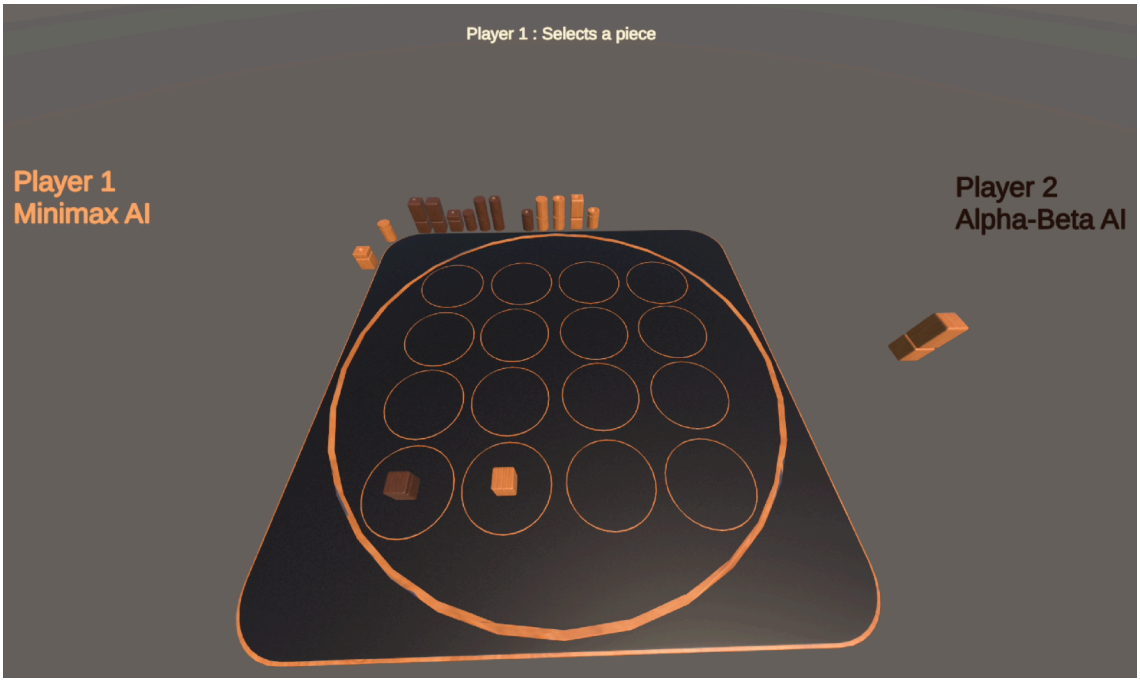

Figure 3.6: Piece Selection Phase

Player 1(Minimax AI) selects one of the remaining pieces and hands it to the opponent(Alpha-Beta AI).

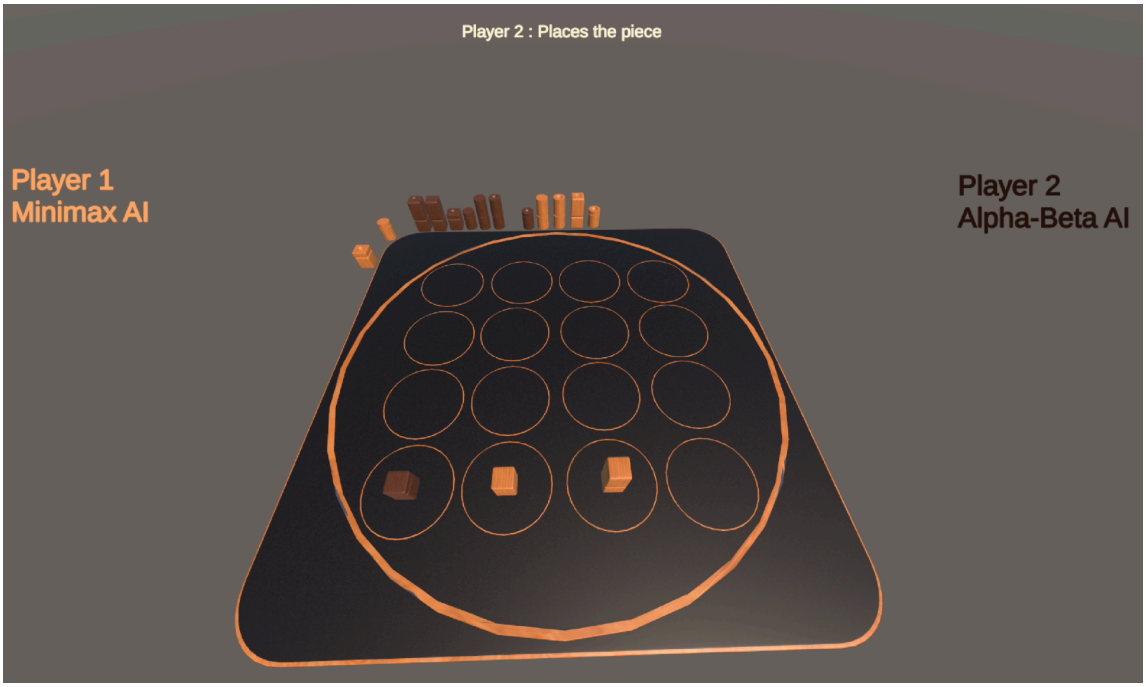

Figure 3.7: Piece Placement Phase

Player2 (Alpha-Beta AI) places the given piece on the board.

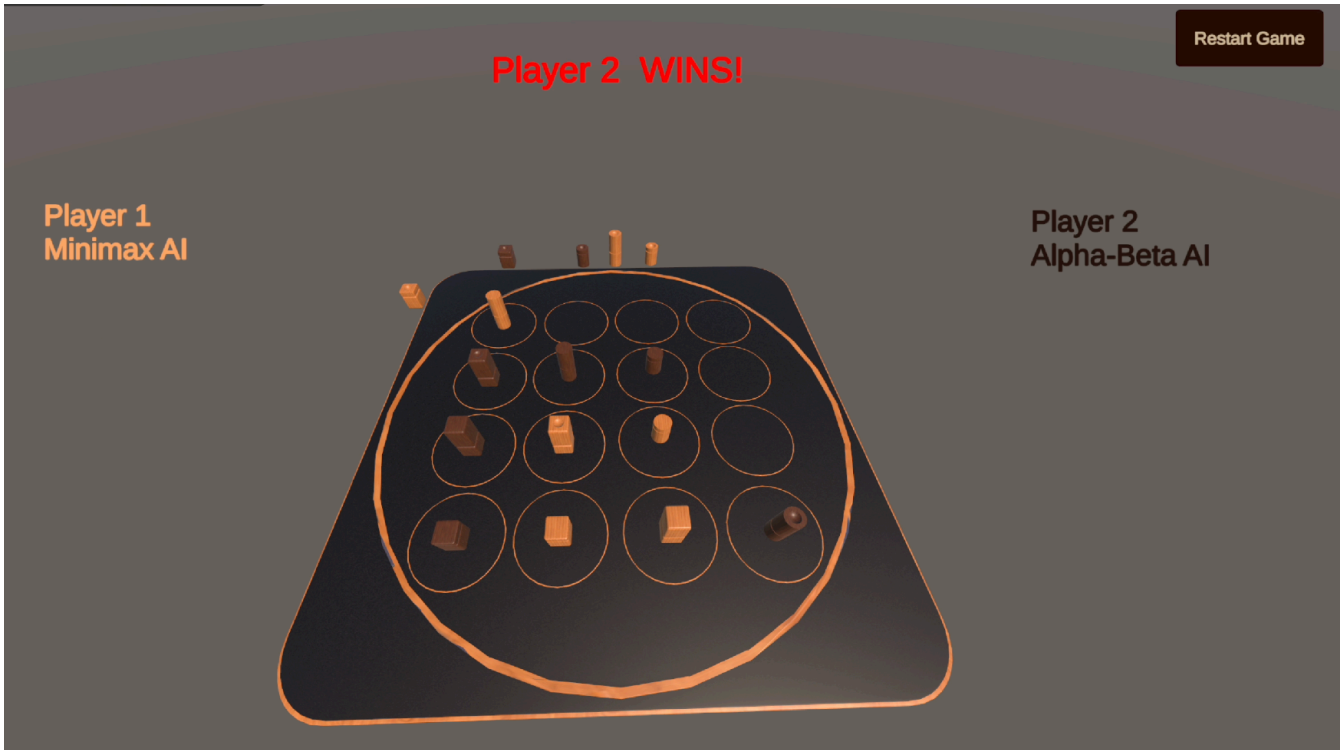

Figure 3.8: Endgame:Win Detection

Player 2 (Alpha-Beta AI) wins the game with a diagonal line of round pieces from top-left to bottom-right. The "Restart Game" button resets the board and pieces for a new match.

Chapter 4
Analysis and Results

4.1 Conducted Tests

To evaluate the performance of Minimax and Alpha-Beta Pruning algorithms using a heuristic evaluation function, I tested them on boards from three game stages. The first stage was the **initial-stage** boards (with 12 to 16 pieces left to play), the second stage was the **mid-stage** boards (with 7 to 11 pieces left), and the third stage was the **late-stage** boards (with only 1 to 6 pieces left to play). For each stage, I took five different boards and ran around 160 unique test combinations for each board. I ran each test 100 times for better accuracy and recorded the number of wins for each player, the number of draws, and the average decision-making time per move. I have recorded and organized all this data in an Excel sheet attached to my report submission.

To keep my work organized, I conducted the tests in two phases:

Phase 1: Self-Play Matches (Intra-agent Testing)- these included self-play matches Minimax vs. Minimax, Alpha-Beta vs. Alpha-Beta and Random vs. Random.

Phase 2: Cross-Play Matches (Inter-Agent Testing)- these included cross-play matches Minimax vs. Alpha-Beta, Alpha-Beta vs. Minimax, Random vs Strategic Agents, Strategic Agents vs Random.

Based on the results in the Excel sheet, I made several observations and drew conclusions about the performance and effectiveness of both algorithms.

4.2 Initial Stage

At the initial stage of the game, when there were still 12 to 16 pieces left to play, the board was mostly open, and there were many possible moves, so players needed to think several steps ahead and plan strategically. That is why this stage was a good test to examine how well each algorithm could deal with many possible options and make smart decisions with limited information.

4.2.1 Self-Play Performance Analysis

Minimax Self-Play

During self-play matches using the Minimax algorithm, I tested decision-making performance across different search depths (from 2 to 7). The results showed a clear pattern in both average decision time and overall gameplay quality. As shown in *Figure 4.2.1*, the average time Minimax took to make a move increased with its depth. At depth 2, it took around 1.83 ms per move, but at depth 5, this increased about 975 ms, and at depth 7, the time reached over 127 seconds per move. This shows that Minimax becomes much slower as it looks deeper into future moves.

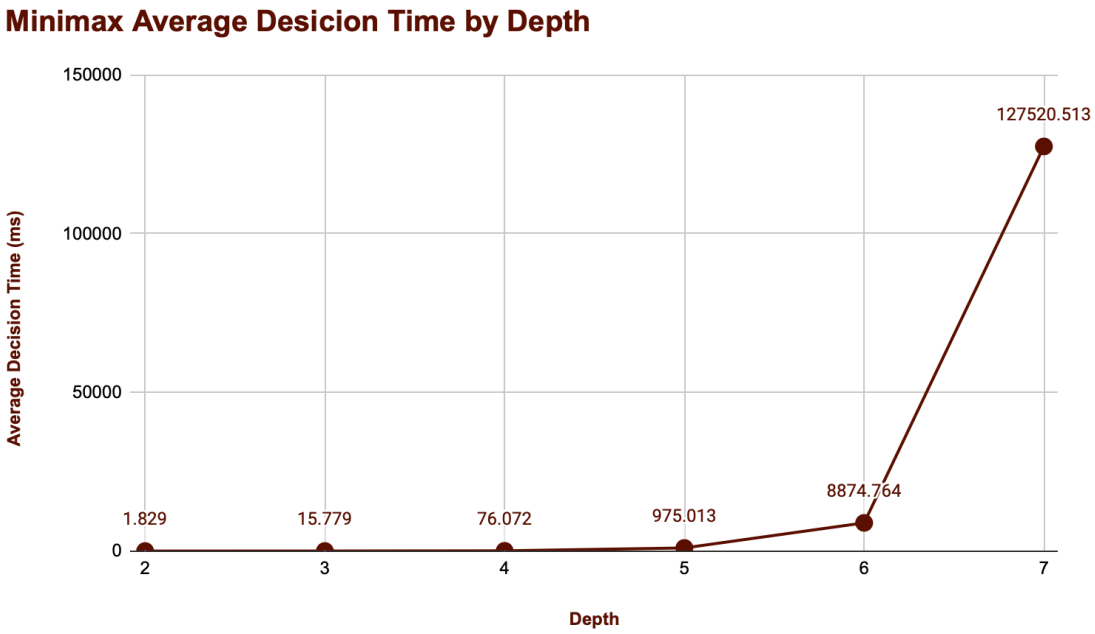

Figure 4.1

Despite the average decision time, deeper search improved minimax performance during the gameplay. As shown in *Figure 4.1*, the win rate increased with each added depth from 30.11% at depth 2 to 71.09% at depth 7. This suggests that Minimax makes much stronger decisions when it has a bigger depth and can look further ahead.

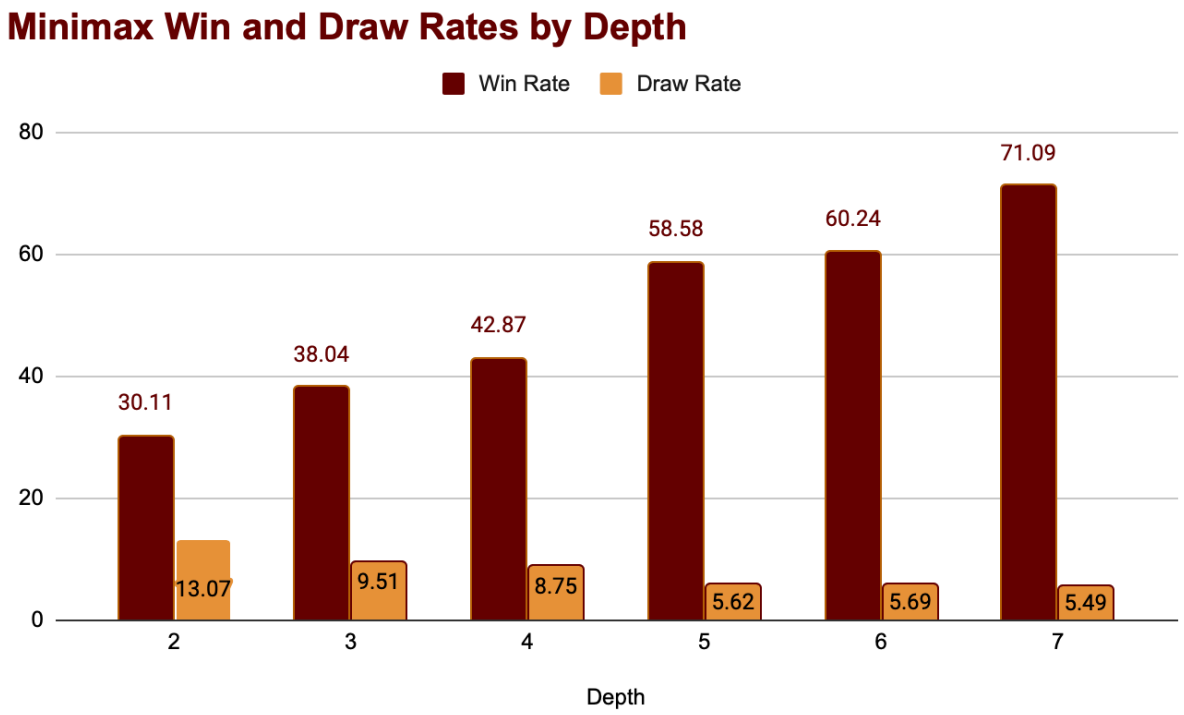

Figure 4.2

Also, in *Figure 4.2* we can see that as depth increased, the number of draws slightly decreased going from 13.07% at depth 2 to 5.49% at depth 7. This shows that Minimax is more likely to find better moves and finish the game with a win when it analyzes more steps in advance.

AlphaBeta Self-Play

The Alpha-Beta Pruning algorithm was also tested in self-play matches across depths 2 to 7. Like Minimax, its performance improved with deeper searches, but compared to Minimax, it made decisions much faster, even in the initial stages of the board.

As shown in *Figure 4.3*, Alpha-Beta made decisions extremely quickly at lower depths. For example, it took just 0.19 ms at depth 2, and even at depth 5, it was still under 13 ms. At depth 7, it increased to about 54 ms, which is still far faster than Minimax at the same depth.

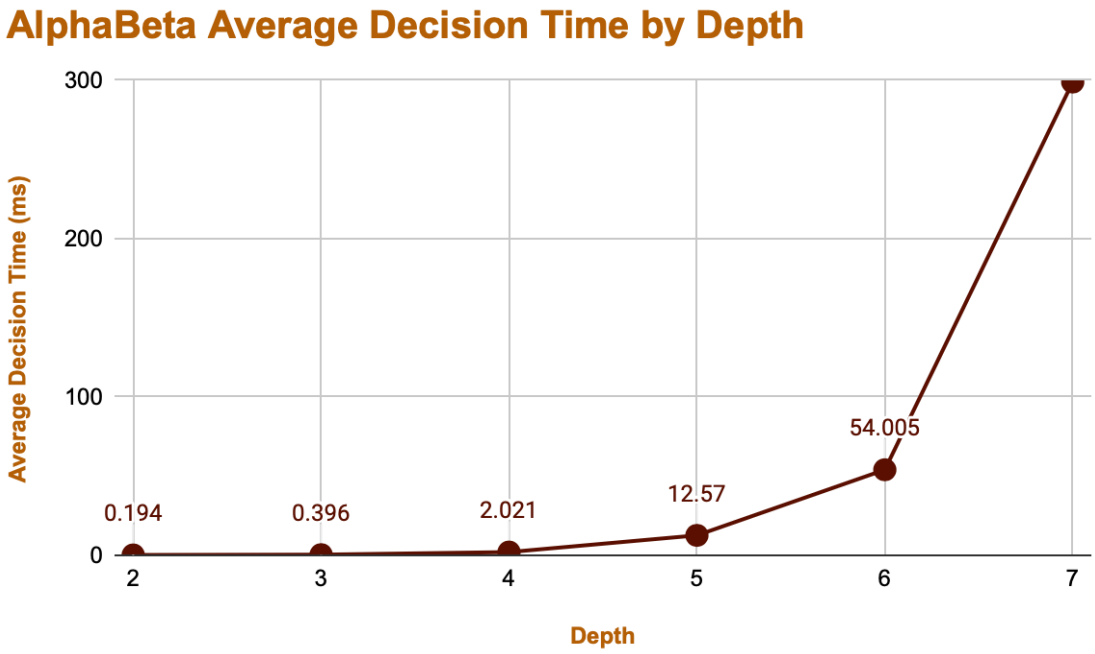

Figure 4.3

As depth increased, Alpha-Beta's win rate improved as shown in *Figure 4.4*. It started at 25.61% at depth 2 and reached 68% by depth 7. This shows that, like Minimax, Alpha-Beta plays better when it can search deeper, but with much less time needed.

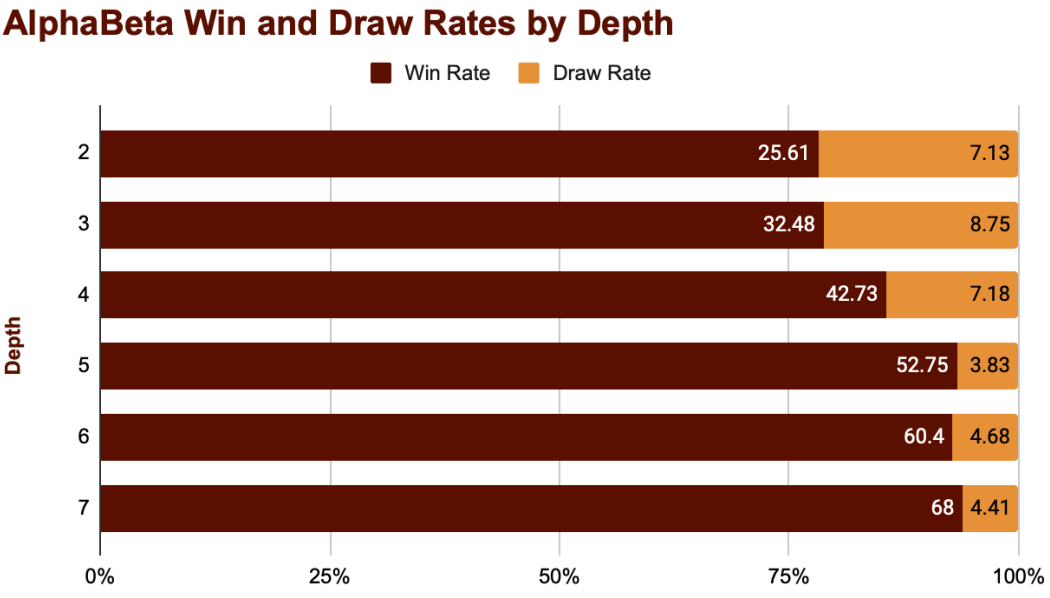

Figure 4.4

4.2.2 Cross-Play Performance Analysis

Alpha Beta and Minimax

To compare the performance of Alpha-Beta Pruning and Minimax, I ran cross-play matches where each algorithm played against the other at different depths. The *Figure 4.5* summarizes their win and draw rates. Across all depths, we can see that Alpha-Beta had a higher win rate than Minimax. At depth 2, Alpha-Beta won 32% of the matches compared to Minimax's 22.56%. We can see the same pattern with higher depths, reaching 64.9% for Alpha-Beta and 27.7% for Minimax at depth 7. This shows that Alpha-Beta was generally more effective during gameplays.

Draw Rates were quite similar for both players and it dropped for both with higher depths, with Alpha-Beta going from 10.3% to 3.8%, and Minimax from 10.2% to 2.033%. This indicates that most matches had a clear winner as depth increased.

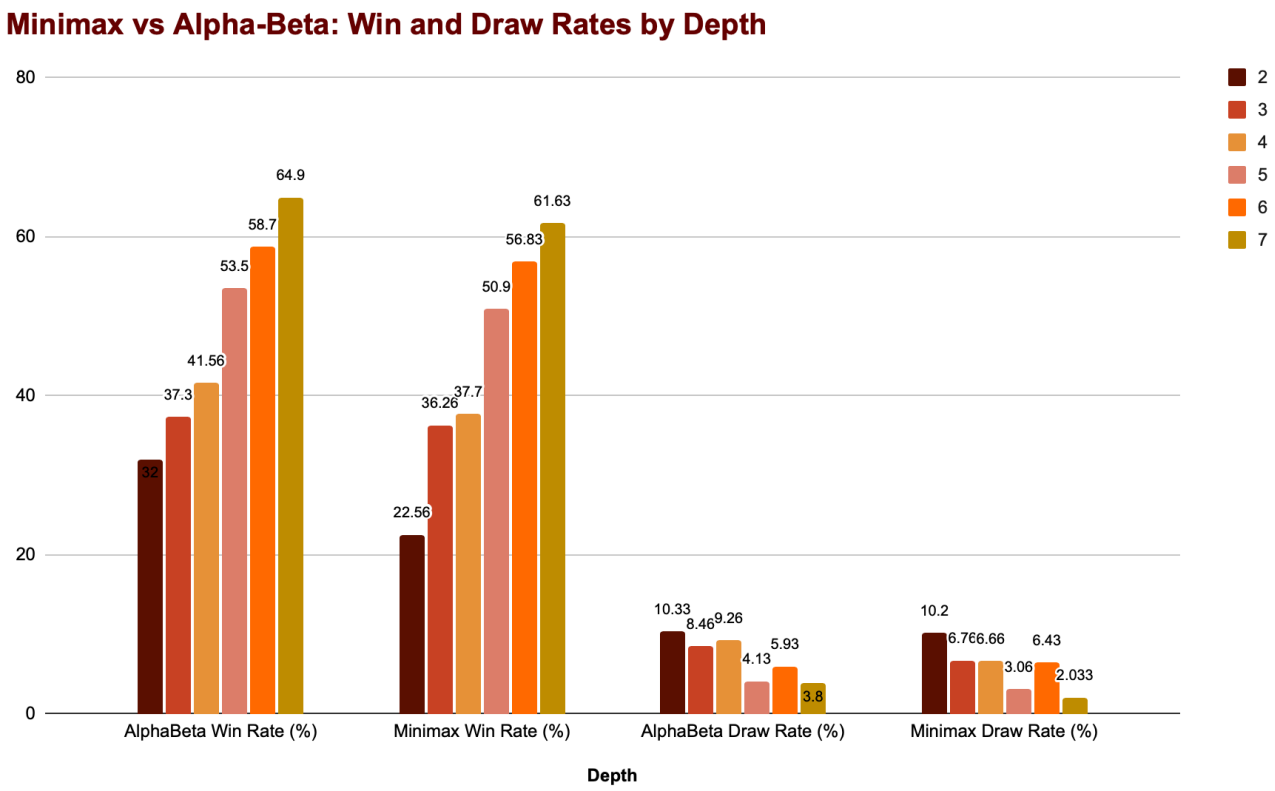

Figure 4.5

Minimax vs Random Players/ Alpha-Beta vs Random Players

Both Minimax and Alpha-Beta clearly outperformed the Random player. Their win rates were high at every depth and got even better as the search depth increased. Alpha-Beta did slightly better overall, especially at the deeper levels. The results are shown in *Figure 4.6* and *Figure 4.7*.

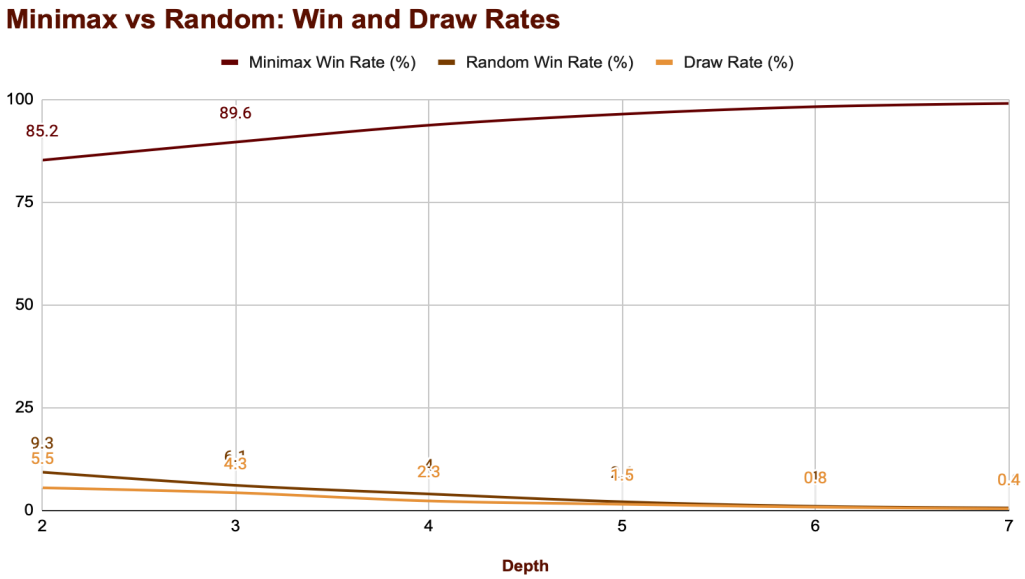

Figure 4.6

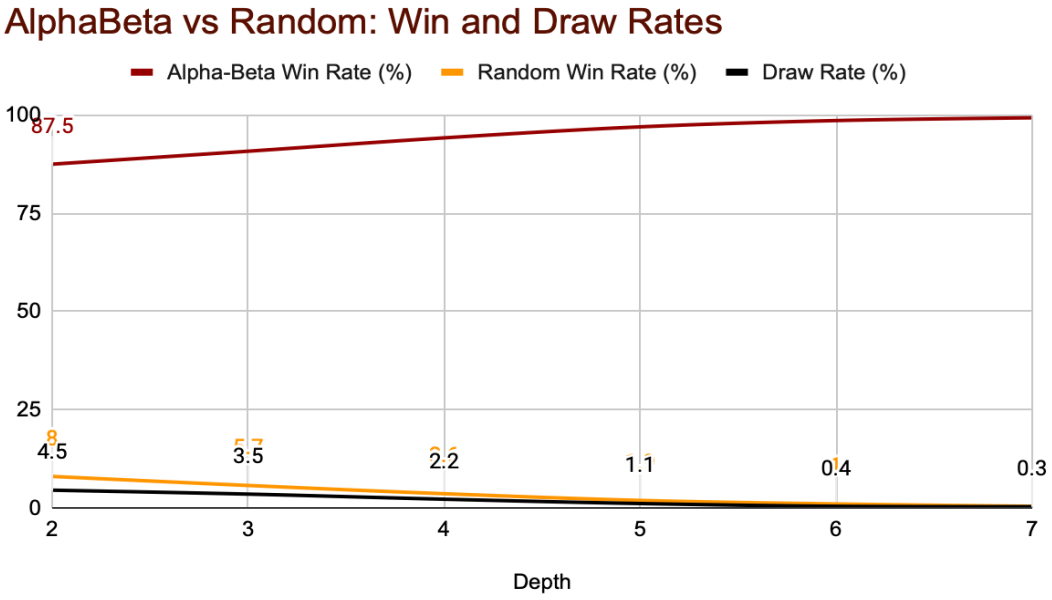

Figure 4.7

4.3 Middle Stage

In the middle stage of the game, when there are about 7 to 11 pieces left, the board is more developed, and there are fewer available moves compared to the early stage. This makes each move more critical. That's why I found this stage necessary for testing how well the algorithms handle tighter situations where it is harder to make good decisions.

4.3.1 Self-Play Performance Analysis

Minimax Self-Play

By looking at winning rates in *Figure 4.8,* we can see that Minimax performed more efficiently than it did in the initial stage, mainly because there were fewer possible moves to evaluate. The chart also shows that the draw rates dropped, likely because the game became more focused, and Minimax could better recognize and complete winning sequences.

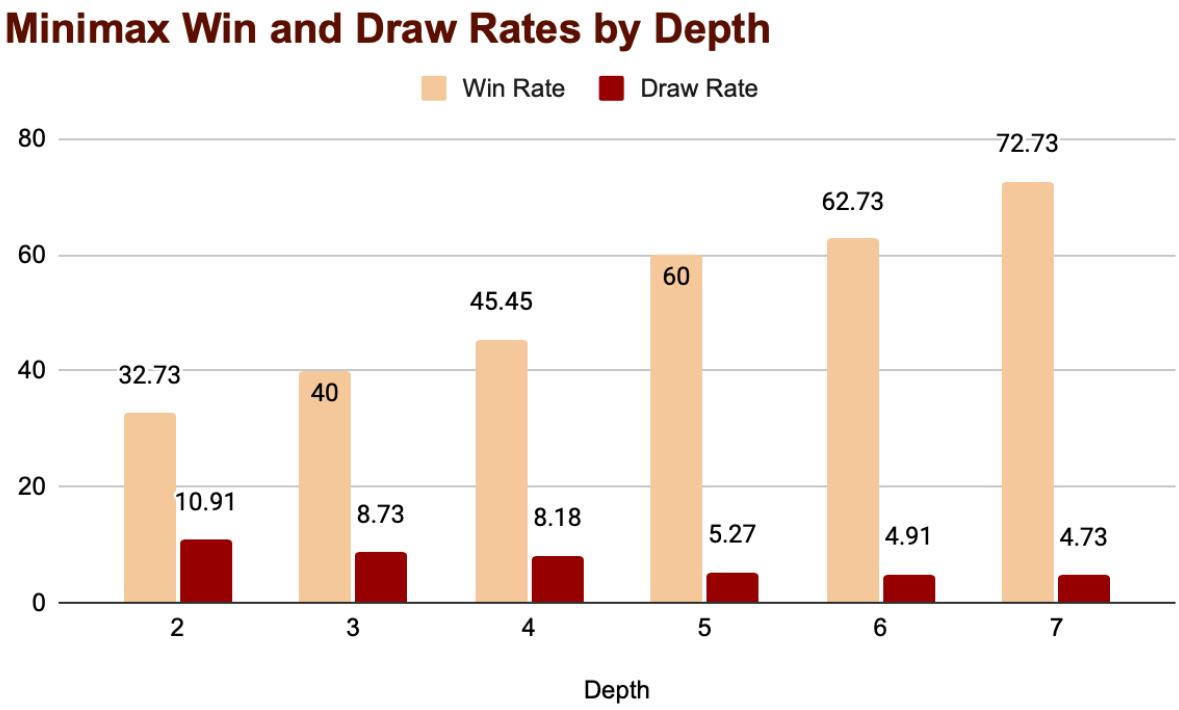

Figure 4.8

Also, in *Figure 4.9*, we can see that the Minimax average decision-making process became slightly faster compared to the initial stage, allowing me to run the program more easily.

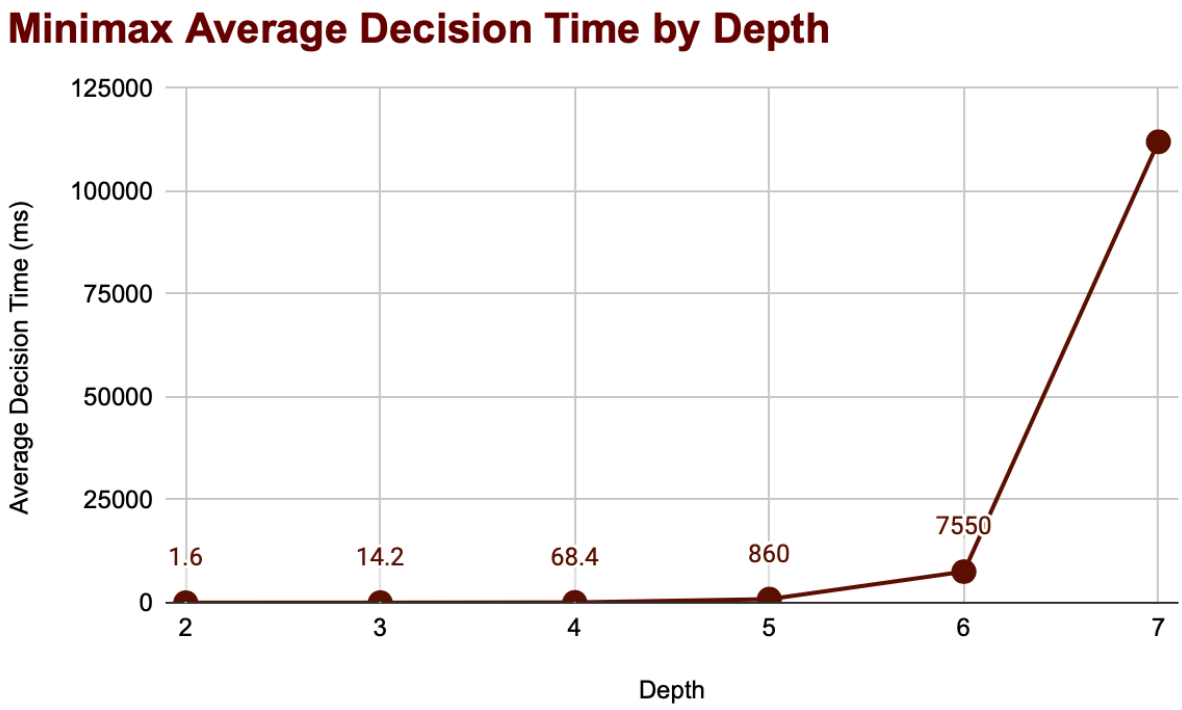

Figure 4.9:

AlphaBeta Self-Play

I saw a similar pattern with the Alpha-Beta player. It also performed more efficiently in the middle stage, with faster decision-making and fewer draws compared to the initial stage. The algorithm handled this phase well, showing strong and stable performance as the number of possible moves decreased. These are shown in *Figure 5* and *Figure 5.1*.

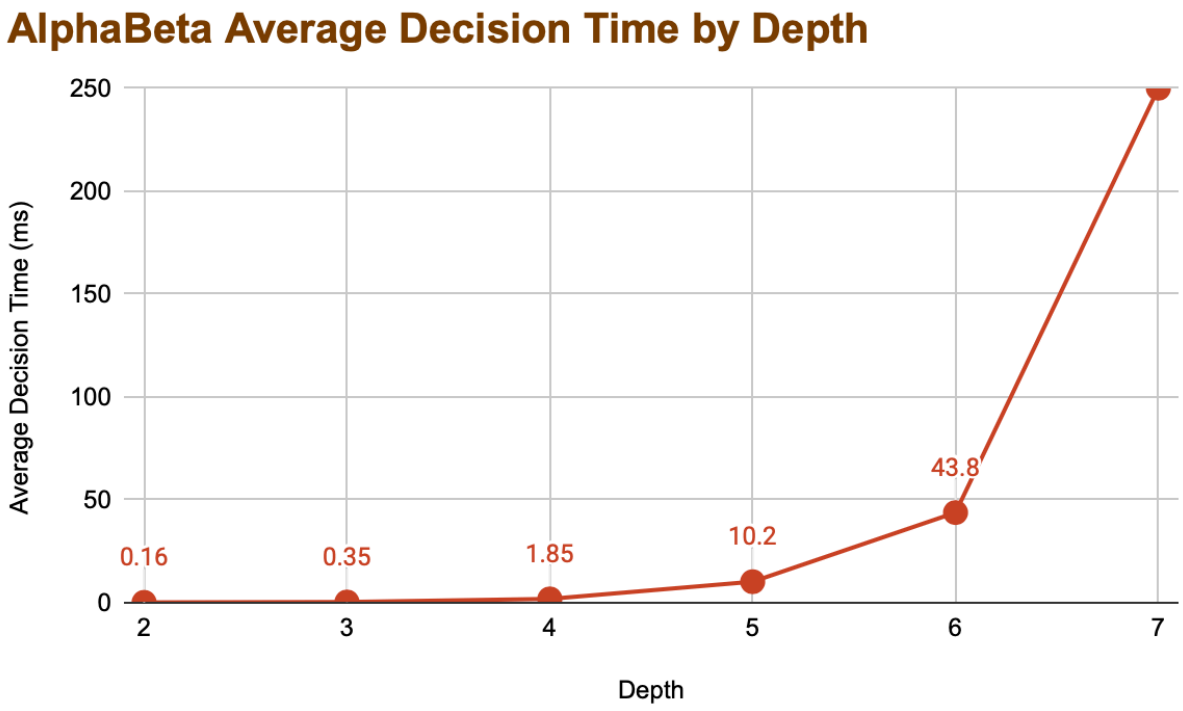

Figure 5.1

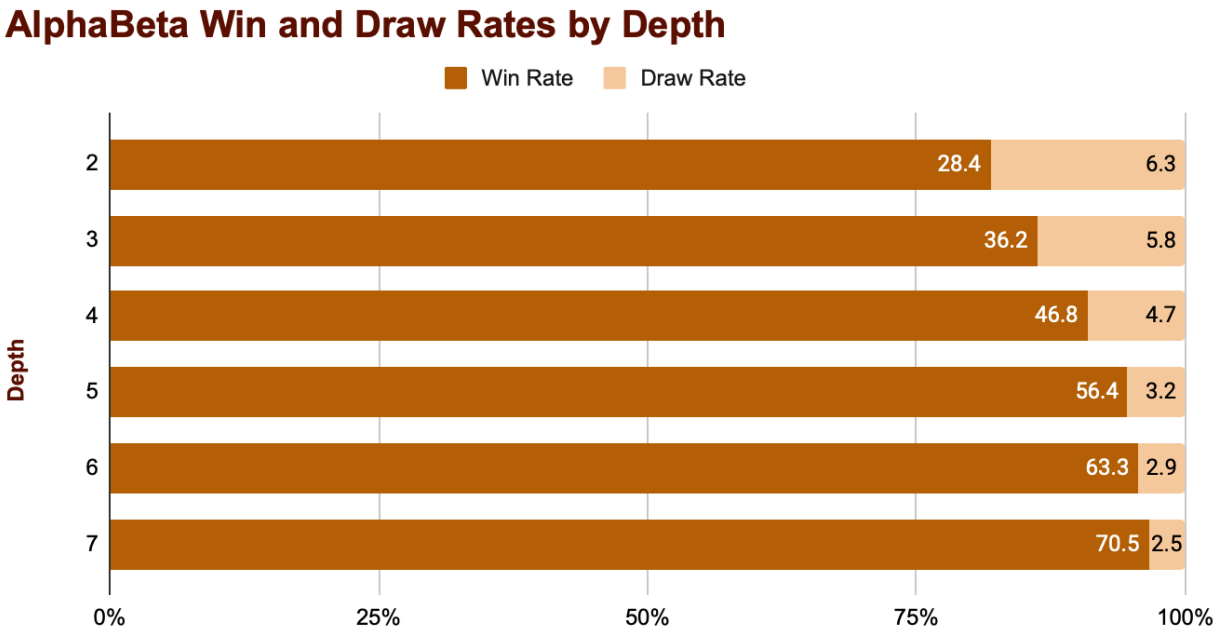

Figure 5.2

4.3.2 Cross-Play Performance Analysis

Alpha Beta and Minimax

In the middle stage, Alpha-Beta and Minimax performed similarly when playing against each other. Compared to the initial stage, the difference between them was not as big, and both handled the game well. This can be seen in *Figure 5.3.*

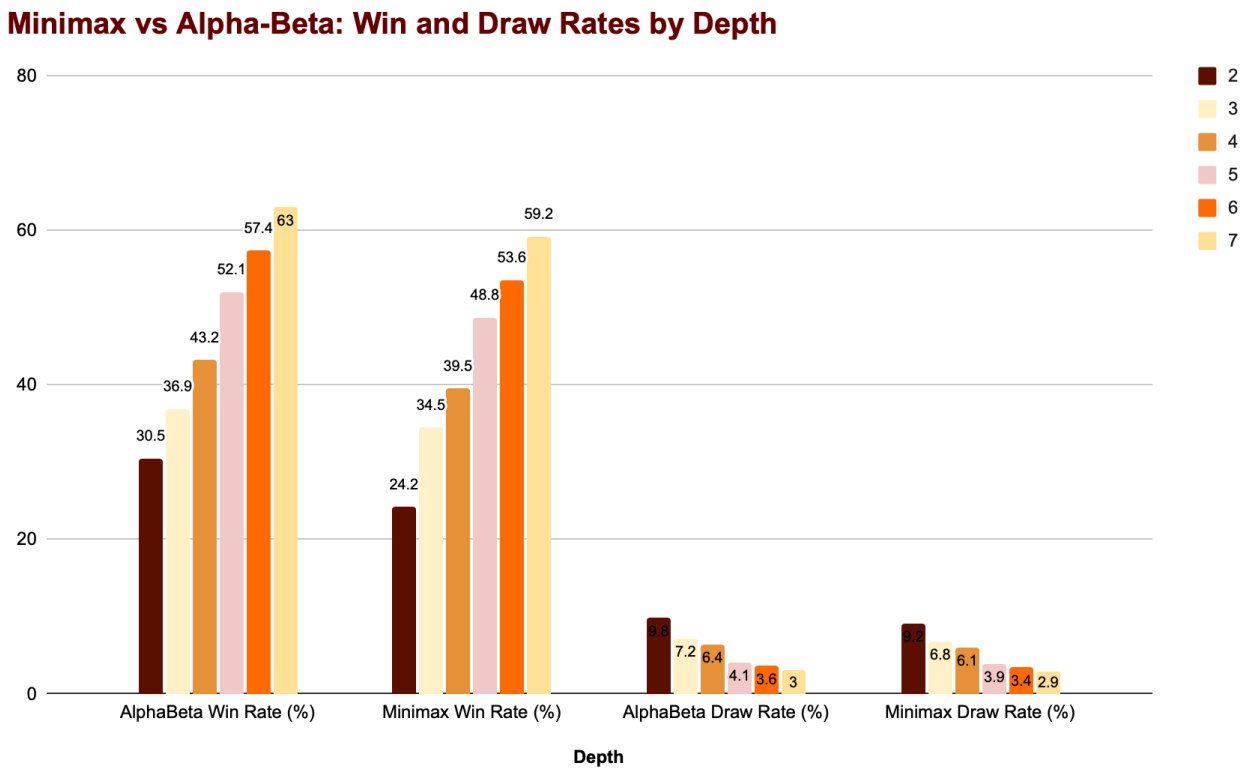

Figure 5.3

Minimax vs Random Players/ Alpha-Beta vs Random Players

In the middle stage, there wasn't much difference between Minimax and Alpha-Beta's performance against the Random player, as both were very strong as shown in *Figure 5.3* and *Figure 5.4*. They won most games, and their performance was almost the same. Also, in this stage, the number of Draws slightly dropped, probably because Random had fewer chances to get lucky with fewer moves left.

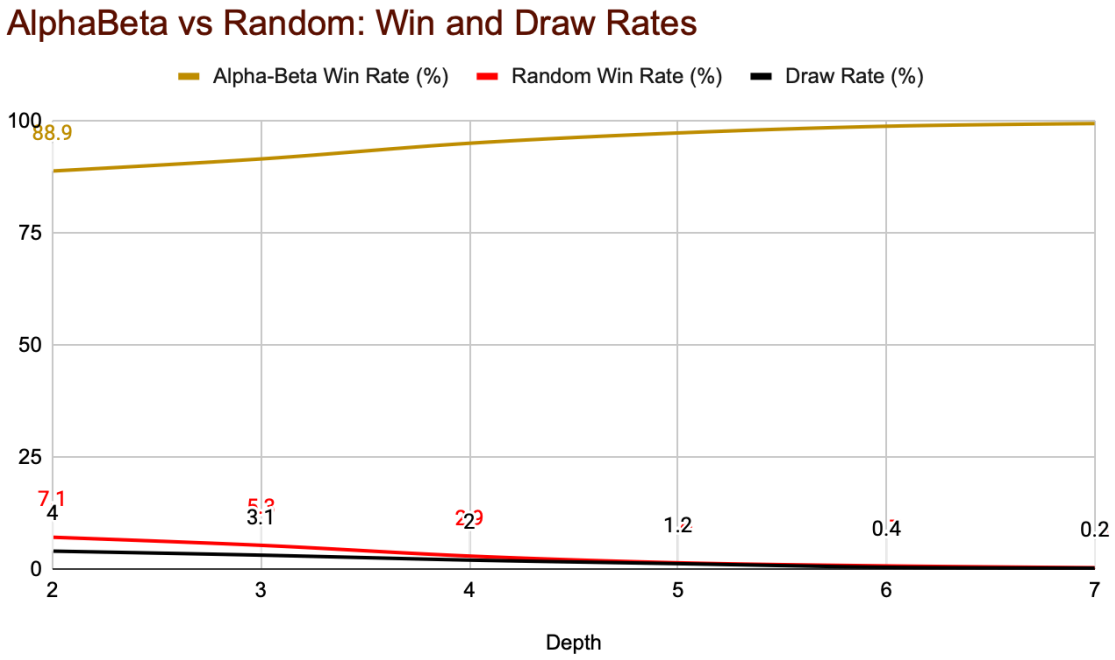

Figure 5.4

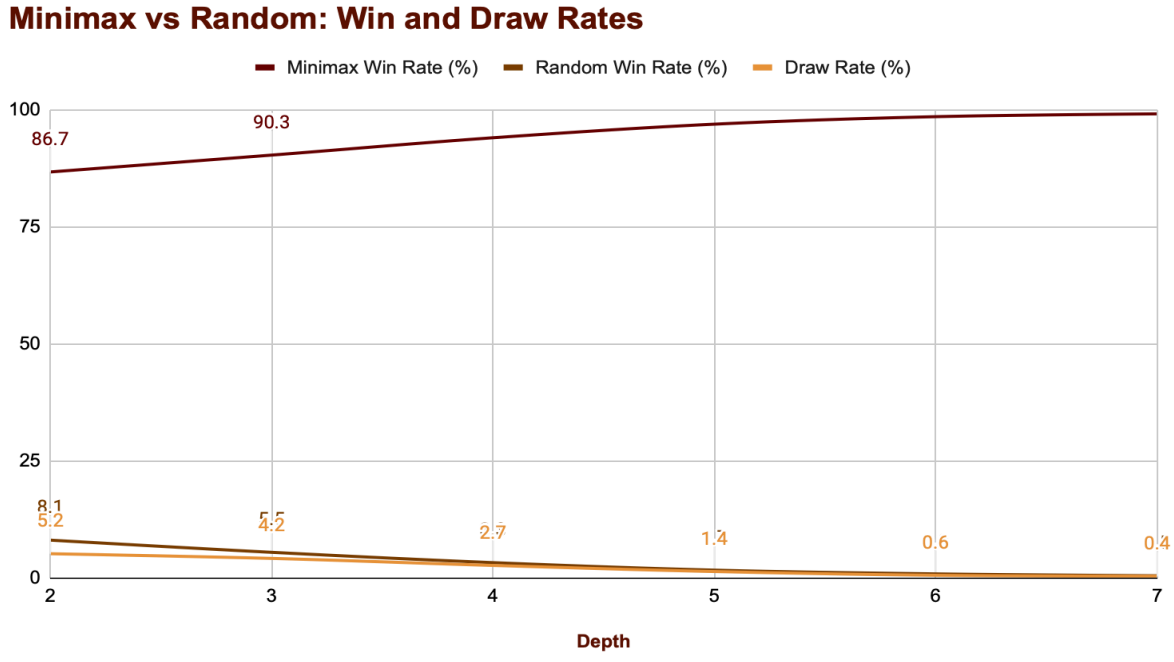

Figure 5.5

4.4 Late Stage

It's worth mentioning that the transition from the middle stage to the late stage followed a similar pattern as the shift from the initial stage to the middle stage. We can see from *Figure 5.6* and *Figure 5.7* that the decision time for both strategic algorithms improved significantly as the number of possible moves became even smaller. At this point, running the program for many matches was very smooth, and both Minimax and Alpha-Beta responded quickly and handled the game efficiently. The Random player had comparably fewer chances to win at this stage since there just wasn't enough space left on the board to get lucky or throw off the other player.

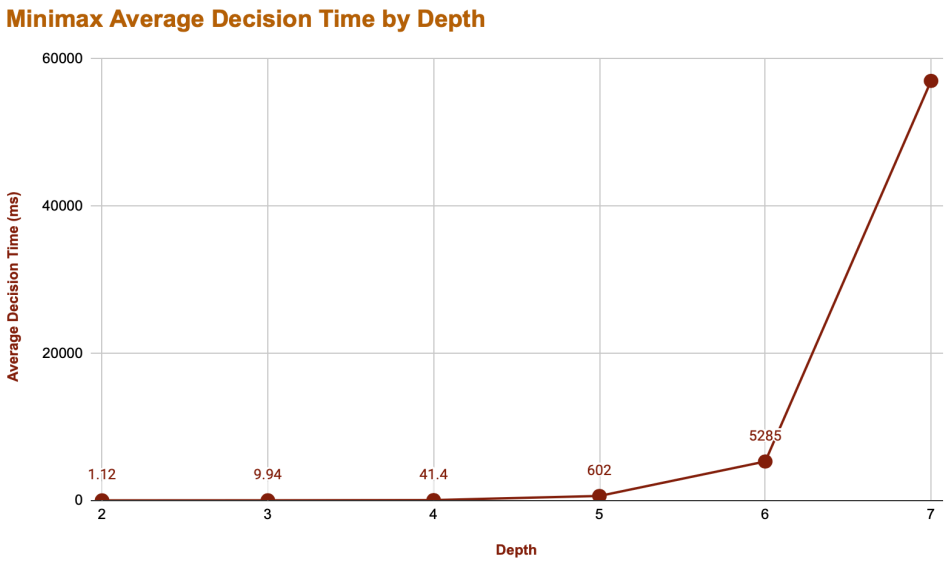

Figure 5.6

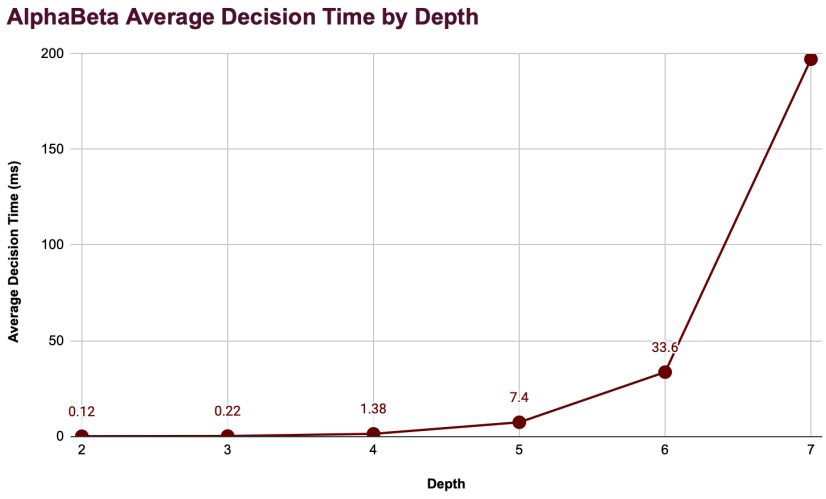

Figure 5.7

4.5 Conclusion and Future Work

Overall, both Minimax and Alpha-Beta performed well across all game stages. Alpha-beta was a bit stronger and much faster, thanks to pruning, especially at deeper levels.

Minimax also gave solid results, just with longer decision times. The difference wasn't huge when they played against each other, especially in the middle and late stages.

Against the Random player, both strategies won most of the time. Random had fewer chances to make lucky moves as the board filled up.

In the future, this project could be extended by adding symmetry-breaking options to reduce redundant board evaluations and improve efficiency. Additionally, incorporating Monte Carlo Tree Search (MCTS) could offer a new direction for handling uncertainty and improve decision-making under more complex conditions.

References

[Figure 1] Quarto – Art of Play. [Online]. Available: https://www.artofplay.com/products/quarto [Accessed: March 18, 2025].

[1] *Quarto Game Rules*, Official Game Rules. [Online]. Available: https://officialgamerules.org/game-rules/quarto/ [Accessed: Apr. 10, 2025].

[2] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed., Global ed. Harlow, United Kingdom: Pearson Education Limited, 2022.

[3] Rowlett, Peter (2015) "Developing Strategic and Mathematical Thinking via Game Play: Programming to Investigate a Risky Strategy for Quarto," The Mathematics Enthusiast: Vol. 12 : No. 1 , Article 9.DOI: https://doi.org/10.54870/1551-3440.1334
Available: https://scholarworks.umt.edu/tme/vol12/iss1/9

[4] D. C. Silva and V. Vinhas, “A flexible extended Quarto! implementation based on combinatorial analysis,” in *Proc. IADIS Int. Conf. Gaming*, Y. Xiao and E. ten Thij, Eds., 2008, pp. 59–66. ISBN: 978-972-8924-63-8.

[5] Jochen Mohrmann, Michael Neumann, and David Suendermann. “An ar-tificial intelligence for the board game ’Quarto!’ in Java”. In: Sept. 2013, pp. 141–146. doi: 10.1145/2500828.2500842.