ANKARA YILDIRIM BEYAZIT UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

INFORMED MONTE CARLO TREE SEARCH FOR BOARD GAMES

M.Sc. Thesis by

Emre YILMAZ

Department of Computer Engineering

May, 2024

ANKARA

INFORMED MONTE CARLO TREE SEARCH FOR BOARD GAMES

A Thesis Submitted to the

Graduate School of Natural And Applied Sciences of

Ankara Yildirim Beyazit University

In Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Engineering, Department of Computer Engineering

by

Emre YILMAZ

May, 2024

ANKARA

M.Sc. THESIS EXAMINATION RESULT FORM

We have read the thesis entitled "**INFORMED MONTE CARLO TREE SEARCH FOR BOARD GAMES**" completed by **EMRE YILMAZ** under supervision of **ASSOC. PROF. DR. FATIH NAR** and we certify that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Fatih Nar

Supervisor

Asst. Prof. Dr. Fadi Yılmaz

Jury Member

Asst. Prof. Dr. Ayşe Nurdan Saran

Jury Member

Prof.Dr. Sadettin Orhan

Director

Graduate School of Natural and Applied Sciences

ETHICAL DECLARATION

I hereby declare that, in this thesis which has been prepared in accordance with the Thesis Writing Manual of Graduate School of Natural and Applied Sciences,

- All data, information and documents are obtained in the framework of academic and ethical rules,
- All information, documents and assessments are presented in accordance with scientific ethics and morals,
- All the materials that have been utilized are fully cited and referenced,
- No change has been made on the utilized materials,
- All the works presented are original,

and in any contrary case of above statements, I accept to renounce all my legal rights.

Date: 2024, 14 May **Signature:** _____

Name & Surname: Emre YILMAZ

ACKNOWLEDGEMENTS

Firstly, I would like to express my sincere gratitude to my advisor Assoc. Prof. Dr. Fatih Nar for his support, comments, and encouragement regarding my thesis.

Beside my advisor, I am deeply indebted to my brother Asst. Prof. Nurullah Yılmaz for his help and endless support.

Finally, I would like to thank my family who supported me throughout my education, showed all their understanding and motivated me.

May, 2024 **Emre YILMAZ**

INFORMED MONTE CARLO TREE SEARCH FOR BOARD GAMES

ABSTRACT

Developing artificial intelligence (AI) agents for adversarial game-playing using search-based methods presents the challenge of creating a robust utility function, which demands significant effort and specialized knowledge. Conversely, hastily devised simple utility functions often produce unsatisfactory outcomes. Monte Carlo Tree Search (MCTS) has emerged as a modern approach that avoids the need for such a strong utility function. Nevertheless, MCTS relies on a substantial number of game simulations to deliver accurate results, incurring notable computational expenses. This study introduces an inventive hybrid approach that leverages MCTS's strengths while seamlessly integrating a modified Upper Confidence Bound for Trees (UCB1) algorithm. This hybridization enhances MCTS's ability to exploit opportunities by including a basic utility function, reducing its reliance on a complex utility function. We conducted a series of experiments, applying this approach to classic board games like Tic-Tac-Toe, Mangala, and English Checkers. These experiments were compared to traditional Minimax and Alpha-Beta Pruning algorithms, along with the pure MCTS method.

Keywords: Adversarial Search, Minimax, Alpha-Beta Pruning, MCTS, Weak Utility

TAHTA OYUNLARI İÇİN BİLGİLENDİRİLMİŞ MONTE CARLO AĞAÇ ARAMASI

ÖZ

Arama tabanlı yöntemler kullanarak rakiplerle oyun oynamak için Yapay Zeka (AI) aracıları geliştirmek, önemli çaba ve uzmanlık bilgisi gerektiren sağlam bir fayda fonksiyonu yaratma zorluğunu ortaya koymaktadır. Aksine, hızlıca tasarlanan basit fayda fonksiyonları sıklıkla tatmin edici olmayan sonuçlar üretmektedir. Monte Carlo Ağaç Arama (MCTS) algoritması, bu kadar güçlü bir fayda fonksiyonuna olan ihtiyacı ortadan kaldıran modern bir yaklaşım olarak ortaya çıkmıştır. Bununla birlikte MCTS, doğru sonuçlar sağlamak için önemli sayıda oyun simülasyonuna dayanır ve bu da önemli hesaplama harcamalarına neden olur. Bu çalışma, MCTS'nin güçlü yönlerinden yararlanırken, değiştirilmiş bir Ağaçlar için Üst Güven Sınırı (UCB1) algoritmasını sorunsuz bir şekilde entegre eden yaratıcı bir hibrit yaklaşım sunmaktadır. Bu hibrit çözüm, MCTS'e temel bir fayda fonksiyonunu dahil ederek fırsatlardan yararlanma yeteneğini geliştirmekte ve karmaşık bir fayda fonksiyonuna bağımlılığını azaltmaktadır. Bu yaklaşımı Tic-Tac-Toe, Mangala ve English Checkers gibi klasik masa oyunlarına uygulayan bir dizi uygulama gerçekleştirdik. Bu uygulamalar, saf MCTS yönteminin yanı sıra geleneksel Minimax ve Alfa-Beta Pruning algoritmalarıyla karşılaştırıldı.

Anahtar kelimeler: Rekabet Ortamında Arama, Minimaks, Alfa-Beta Budama, MCTS, Basit Fayda Fonksiyonu

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Artificial intelligence is typically defined as a computer system's ability to perform mental processes such as reasoning, understanding, and learning from past experiences, which are often characteristic of human nature. Currently, AI is used effectively in numerous fields. Examples include self-driving cars in the automobile industry, disease detection and treatment planning in healthcare, autonomous aircraft in defense, as well as risk assessment and investment advice in finance. In addition, artificial intelligence tools such as voice assistants, computer games, recommendation systems, and social media algorithms have taken their place in daily life. On the other hand, AI is used in gaming and decision-making problems. The most important artificial intelligence technique in this process is adversarial search. Adversarial search is a problem-solving technique that focuses on the decision-making process in competitive or contentious situations. This method is used to find optimal strategies in scenarios where multiple players have opposing or conflicting goals. Players try to determine the best moves by taking into account their opponents' possible moves and countermoves. This technique is widely applied in board games like checkers, go, and chess, allowing AI agents to think and outmaneuver opponents.

Board games have become an important topic in artificial intelligence research as they are an important testing environment for artificial intelligence development methods. Deep Thought was an early computer that managed to defeat a chess champion in the 1980s. IBM Deep Blue II systems achieved victory against world chess champion Kasparov in 1997, creating a significant impact globally [1]. More recently, Google DeepMind's AlphaGo program, written in 2015, defeated Fan Hui at the 2nd Dan level in October 2015 and Lee Sedol at the 9th Dan level in March 2016, clearly demonstrating artificial intelligence's success in games [2].

Board games are generally a type of game that involves moving pieces on a table or on a specific playing field according to certain rules. Board games have a lengthy and vibrant history that can be traced back thousands of years. Senet, which originated in

ancient Egypt around 3500 BC, is considered the oldest known board game [3]. They are games of skill, strategy, and chance played by two or more people. Games such as chess, checkers, monopoly, scrabble, and backgammon can be given as examples. Board games are important not only for entertainment purposes but also for algorithm development.

There are many methods for conducting adversarial searches. Minimax and Alpha-Beta Pruning are the most well-known methods. These methods involve finding the appropriate move by examining all possible moves against each move of the opponent in two-player, zero-sum games. In games where the branching factor is high, it becomes impossible to examine all movements. In these cases, utility functions that best express the general state of the game are needed. However, there are methods that can work without developing any utilities. One of the known adversarial search methods that works without a utility function is Monte Carlo Tree Search (MCTS).

MCTS is an approach that constructs a search tree through random sampling within the decision space to pinpoint the best decisions within a given domain. This method strives to determine the most favorable choices within a specific domain without the need for a positional evaluation function. Essentially, MCTS is a Best First Search (BFS) method that leverages the outcomes of random simulations to guide decision-making. In addition, this algorithm employs a probabilistic approach, utilizing quick random simulations to strategically expand a game tree. This method is especially effective in areas with extensive search spaces, areas where deterministic algorithms have traditionally struggled.

MCTS extracts statistics by sequentially applying the selection, expansion, simulation, and backpropagation steps over a period of time. This algorithm has gained popularity in various domains, including games like Go, due to its ability to handle hidden information and uncertainty effectively. The MCTS has been described as a groundbreaking algorithm in the realm of artificial intelligence.

The MCTS algorithm has various applications beyond games and is effective in

scenarios characterized by state-action pairs, employing simulations to assess action quality. Its versatility is demonstrated in diverse areas, notably excelling in complex strategic games like Go, chess, shogi, and poker, often outperforming human capabilities.

1.1 Problem Statement

1.2 Objectives

In this thesis, we study the MCTS algorithm variant for combinational games, which are a specific category of games that involve two players and adhere to certain fundamental principles [4]. We focus mainly on board games like Checkers, $N \times N$ Tic-Tac-Toe, and Mangala. The proposed MCTS variant is based on a hybridization of the MCTS concept with weak utility functions.

1.3 Scope and Limitations

MCTS is an algorithm that balances between exploring new possibilities and exploiting known ones by conducting random simulations and tracking the outcomes of actions to make informed decisions iteratively. It has been effectively used in areas with large search spaces and combinatorial games. Nevertheless, MCTS has drawbacks, including a reliance on the model's assumptions and simplifications. In complex games with many possible moves or limited time, the successful use of MCTS often requires tailored adjustments and combinations with other methods. This applies not only to gaming but also to various real-world applications such as physics simulation, scheduling tasks, and security. The approach has been very effective in perfect information games, but there is now an ongoing study into how well it works in games with ambiguity and hidden information. Consequently, even though MCTS is a strong algorithm, its use is limited to domains with large search spaces, and it has some drawbacks, such as the requirement for problem-specific adjustments and its reliance on underlying assumptions and simplifications.

CHAPTER 2

LITERATURE REVIEW

2.1 Overview of Artificial Intelligence (AI)

AI is a rapidly developing discipline that has gained a lot of interest in recent years. AI encompasses the creation of computer systems and algorithms that can perform tasks usually requiring human intelligence, such as learning, problem-solving, decision-making, and language comprehension. Notable academics such as Alan Turing, John McCarthy, and Marvin Minsky initiated investigations into the feasibility of developing machines capable of human-like thinking and reasoning during the 1950s, marking the inception of AI. As a result, the domain of artificial intelligence has experienced significant progress, propelled by the rapid increase in computational capacity, the availability of extensive datasets, and the creation of intricate algorithms and methodologies. A diverse range of industries and domains, including healthcare, banking, transportation, education, and entertainment, are widely adopting and applying AI. Machine learning, natural language processing (NLP), computer vision, and robotics are prominent domains in AI study and development.

Machine learning is a specialized area in artificial intelligence focused on developing algorithms and statistical models. These models enable computer systems to improve their performance by analyzing data, learning, and adapting without requiring direct programming instructions. These approaches encompass supervised learning, unsupervised learning, and reinforcement learning [5].

NLP, a subset within the realm of AI, concerns itself with the interplay between computers and human language, encompassing tasks such as understanding, generating, and processing natural language text or speech. NLP encompasses the creation of algorithms and models capable of comprehending, interpreting, and generating human language. This enables the development of applications like virtual assistants, linguistic conversion, and emotion detection [6].

Computer vision is a specialized area within artificial intelligence focused on developing algorithms and models adept at analyzing and understanding digital images and videos. These tasks encompass object detection, picture categorization, and facial recognition [7].

Robotics is the integration of AI with physical systems, enabling the development of self-determining or partially self-determining machines that can fulfill a broad spectrum of tasks, from manufacturing to healthcare [8].

In conclusion, AI is a dynamic and rapidly progressing field that carries significant implications for society, industry, and academia. Its ongoing development offers the potential to tackle intricate problems and open up new possibilities in diverse sectors.

2.2 Historical Perspectives on AI in Gaming

The incorporation of AI in video game design has been fundamental since the inception of gaming, evolving alongside technological advancements in the industry. Understanding the historical context of AI in gaming is critical to evaluating the current state of AI in gaming and its potential future developments.

The earliest use of AI in gaming can be traced back to the 1950s and 1960s, with the development of chess programs that could play at a high level. These programs used rule-based systems and decision trees to make moves based on the present condition of the game. In the 1970s and 1980s, AI was used in arcade games and early console games like Pac-Man and Space Invaders. These games used simple AI algorithms that could generate random behavior or follow pre-defined patterns [5, 9].

In the 1990s and early 2000s, the emergence of machine learning and deep learning techniques caused a significant shift in applying artificial intelligence to games. These techniques allowed AI agents to learn from their environment and adapt to player behavior, creating more sophisticated and dynamic gameplay experiences. For example, in the game *Creatures*, AI agents were developed to learn and evolve over time, allowing players to engage with them realistically. In recent years, AI has been used in game

development and design, allowing developers to create riveting and interactive game worlds. AI algorithms can be used to generate terrain, create non-playable characters (NPCs) with unique personalities and behaviors, and simulate complex systems such as weather and day/night cycles [4, 10, 11].

Nowadays, AI plays a crucial role in contemporary gaming, being utilized across various aspects including game development, design, player interaction, and gameplay. AI algorithms can be utilized to generate NPCs that can learn from player behavior and adapt their strategies accordingly, creating more challenging and engaging gameplay experiences. However, there are still challenges and limitations to the current AI in gaming. For example, AI agents often struggle to generalize from one game to another, limiting their ability to transfer their learning to new environments. Additionally, creating AI that is capable of comprehending and reacting to human emotions is a significant challenge, as human emotions are complex and often difficult to quantify [12, 13].

The historical development of AI in gaming has been marked by significant advancements in technology and techniques. From the early use of rule-based systems and decision trees to the introduction of machine learning and deep learning techniques, AI has transformed the gaming industry, allowing for the creation of more immersive and interactive game worlds. However, current AI in gaming still faces challenges and limitations, and the potential for future AI developments is significant. With the continued advancement of AI technology, we can anticipate witnessing even more advanced and dynamic gameplay experiences with AI agents that can learn from their environment, adapt to player behavior, and understand human emotions. These developments will have significant implications for the gaming industry as well as for the broader field of AI and its applications in other domains [14].

2.3 Overview of Adversarial Games

Adversarial games, which are also called competitive games or zero-sum games, involve strategic interactions where multiple players compete against each other with opposing

objectives. In such games, when one player gains, it directly translates to losses for the other player(s), leading to an outcome where gains and losses balance out to zero. Adversarial games have received considerable attention in disciplines like game theory, computer science, and artificial intelligence. They offer a structured approach to understanding and evaluating a variety of real-world situations, including economics, cybersecurity, military tactics, and political decision-making [15]. One of the fundamental concepts in adversarial games is the Nash equilibrium, which represents a stable state where no player can enhance their outcome by individually modifying their strategy. Finding the Nash equilibrium is a crucial step in understanding the optimal strategies for players in an adversarial game [16]. Another important aspect of adversarial games is the concept of Minimax, which is a decision-making rule that aims to minimize the maximum possible loss. This principle is widely used in game-playing algorithms, such as the famous Minimax algorithm in chess and other board games [5].

Adversarial games have also been examined within the realm of machine learning, where they have been used to develop techniques like generative adversarial networks (GANs) and adversarial training. These methods leverage the competitive nature of adversarial games to enhance the performance and robustness of machine learning models [11]. Overall, the study of adversarial games has contributed significantly to our understanding of strategic decision-making, conflict resolution, and the development of intelligent systems.

2.4 Approaches in Adversarial Games

2.4.1 Algorithmic (Search) Approaches

Adversarial games present a challenge in the field of AI and decision-making because they involve multiple agents competing against one another with the goal of maximizing their own utility. This subchapter will provide an overview of various search methods for solving adversarial games, including the Minimax algorithm, Alpha-Beta Pruning, Expectimax, and MCTS.

The Minimax algorithm is a classic approach for solving adversarial games, which involves recursively exploring the game tree and selecting the move that minimizes the maximum possible loss for the agent [5]. The game tree analyzed with the Minimax algorithm for the Tic-Tac-Toe game is exhibited in 2.1. The algorithm ensures the best solution in a two-player zero-sum game, yet its computational demands increase exponentially with the game's complexity, rendering it unfeasible for extensive and intricate games due to its exponential growth in computational complexity with the game's depth.

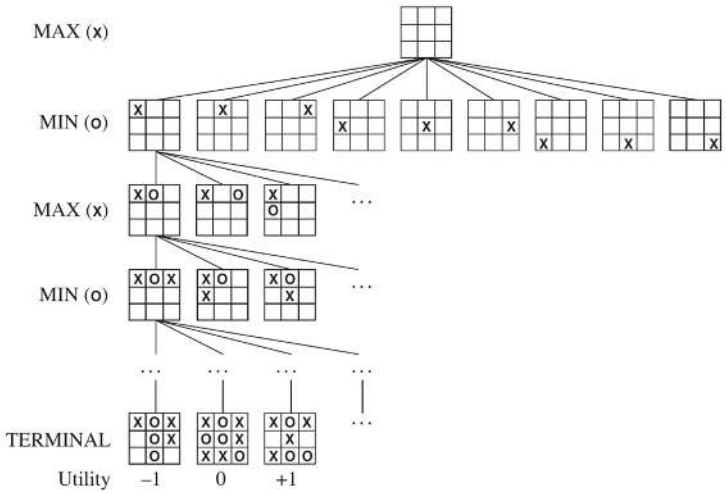


Figure 2.1 Minimax game tree representation of Tic-Tac-Toe

Alpha-Beta Pruning is a method that enhances the efficiency of the Minimax algorithm by removing branches of the game tree that are certain to have a lower utility than the current best choice. This technique decreases the number of nodes that need to be explored, resulting in a notable reduction in computational complexity without affecting the optimality of the solution [17].

Expectimax is a generalization of the Minimax algorithm that can be used to solve games with more than two players or games with stochastic elements. In this approach, the algorithm calculates the expected value of the utility for each possible action, taking into account the probabilities of the different outcomes [18]. This strategy is especially valuable in games characterized by incomplete information, where the agent lacks comprehensive knowledge of the game's current state.

MCTS is a strategy that employs simulated exploration of the game tree, leveraging random sampling to navigate towards favorable paths based on the outcomes observed during the search process [4]. This approach is especially beneficial for games with extensive and intricate game trees, where the computational demands of the Minimax algorithm make it impractical to use. MCTS has been implemented with success in a variety of games like Go, poker, and video games.

In conclusion, the field of adversarial games presents unique challenges and opportunities for artificial intelligence research. The Minimax algorithm, Alpha-Beta Pruning, Expectimax, and MCTS are just a few of the methods that have been developed to solve these games. As the complexity and scale of adversarial games continue to grow, it is likely that new and innovative approaches will be needed to tackle these challenges.

2.4.2 Data-Driven Approaches

Data-driven approaches in adversarial games refer to techniques that leverage data and machine learning to model the interactions between the adversary and the defender (e.g., a learning system) in an adversarial setting. Supervised learning, unsupervised learning, and transfer learning methods can be considered as data-driven approaches used in adversarial games [19].

Supervised learning techniques such as classification, regression, and neutral networks are machine learning methods that generate a pattern or function from labeled training data based on past experiences. The created model can be used to predict the most appropriate action given the current game state, capturing complex patterns and heuristic methods that are difficult to manually code.

Unsupervised learning is a machine learning technique used to extract patterns or relationships from unlabeled datasets. In this technique, there is no predetermined output or label for the data. Instead, the algorithm attempts to discover structures in the dataset and provides information about these structures. It can be used to identify patterns that may be useful in decision-making in adversarial games and to extract

features from game data.

Reusing information from one task or domain to enhance performance on a related task or domain is known as transfer learning. Transfer learning may be employed in adversarial games to adjust models that have been trained on one game to achieve high performance on a different, yet related, game. This can be especially advantageous when the desired game has a scarcity of training data accessible [20].

2.4.3 Reinforcement Learning

Reinforcement learning can be a valuable approach when dealing with adverserial search problems where training data is not available and supervised learning cannot be used. In such cases, reinforcement learning, a type of machine learning that relies on rewards and punishments to train agents, may be a suitable alternative. This method allows systems to learn and improve based on their own actions and experiences in interactive environments without the need for labeled datasets. Using reinforcement learning, systems can autonomously explore and discover the actions that lead to the most reward; this makes it a potent tool for tackling complex search problems where traditional search algorithms may struggle due to the complexity of the task.

Deep reinforcement learning, known as the integration of deep neural networks and reinforcement learning, has revolutionized adversarial game-playing. Deep neural networks can be used to learn powerful representations of the game state, while reinforcement learning algorithms like Q-learning or policy gradients can be used to learn optimal policies.

Policy gradient methods constitute a subset of reinforcement learning algorithms focused on directly enhancing the parameters of a policy function. This policy function is in charge of mapping game states to the actions that should be taken. These approaches can be particularly effective in adversarial games where the reward signal is sparse or delayed, as they can learn to make decisions based on long-term outcomes in place of immediate rewards.

2.5 Gaps and Challenges in Existing Approaches

Despite the various methods developed for solving adversarial games, there remain deficiencies and hurdles within current methodologies.

The Minimax algorithm, while effective, can become computationally expensive as the depth of the game tree increases. This is because the algorithm explores all possible outcomes, which can lead to exponential growth in the number of nodes to be assessed [5].

Alpha-Beta Pruning is an adaptation of the Minimax algorithm aimed at decreasing computational costs by pruning branches that will not affect the final result. Unlike the Minimax algorithm, parameters called alpha and beta are stored in each node. These parameters help decide whether to visit certain branches. However, the effectiveness of Alpha-Beta Pruning depends on the sequence in which the nodes are assessed. If nodes are assessed in a suboptimal order, the algorithm may fail to prune effectively, leading to increased computational costs [21].

Expectimax is designed for games with stochastic elements, but it assumes that the probability distributions are known and stationary. However, in many real-world scenarios, the probability distributions may be unknown or non-stationary, which can lead to suboptimal solutions or failure to converge to a solution [22]. Moreover, since this algorithm is based on the Minimax algorithm, it faces the same disadvantages. On the other hand, in cases with a high branching factor, it is not possible to evaluate every move using Minimax-based methods. In these situations, utility functions that best express the general state of the game are needed. However, it is not easy to determine a utility function suitable for every problem and every situation. To solve this problem, the MCTS algorithm, which does not need any utility functions, comes into prominence.

MCTS is a simulation-driven method that has achieved success in many fields, like board games and video games. Nevertheless, MCTS encounters the difficulty of effectively managing the trade-off between exploring new possibilities and exploiting

existing knowledge throughout the search process. This necessitates conducting a substantial number of simulations or rollouts in order to gather precise data and make well-informed choices [23].

In summary, while existing approaches have been successful in solving adversarial games, there are still gaps and challenges that need to be considered. These include computational cost, node ordering, partial observability, and sensitivity to simulation policy. Addressing these challenges will require further research and the development of novel approaches that can handle the complexity and uncertainty of adversarial games.

CHAPTER 3

OVERVIEW OF THE RELATED METHODS

3.1 Minimax

The Minimax strategy is widely recognized as the primary method for playing two-player, zero-sum games. In 1928, John von Neumann provided the proof for the Minimax theorem [16]. Minimax involves a consistent approach aimed at minimizing the maximum potential loss resulting from a player's decisions.In other words, when applying the Minimax algorithm, a player evaluates all possible outcomes based on their opponent's best responses to their own strategies. The player then chooses the strategy that maximizes the potential payoff when considering the best possible response from the opponent. The Minimax algorithm simply tries to minimize the loss and, therefore, maximize the gain.

The algorithm involves two players, where one is referred to as the maximizing player and the other is named the minimizing player. Each player selects the optimal move for themselves. The maximizing player tries to get the highest score possible, while the minimizing player tries to do the opposite and get the lowest score possible.

Minimax generates a game tree that encompasses all potential moves of both the player and the opponent, starting from the initial state and extending to the conclusion of the game. After reaching the conclusion of the game, the outcome is determined using utility functions. These functions are based on the principle of zero-sum games, where the outcome is structured so that any increase in one player's score directly translates to a decrease in the other player's score, maintaining a constant total score of zero. This setup ensures that one player's success hinges on the other player's failure, creating a scenario where victory for one necessitates defeat for the other.

In Figure 3.1, leaf nodes are assigned scores based on the utility function. The green path in the illustration represents the best moves for each player at various positions in the game.

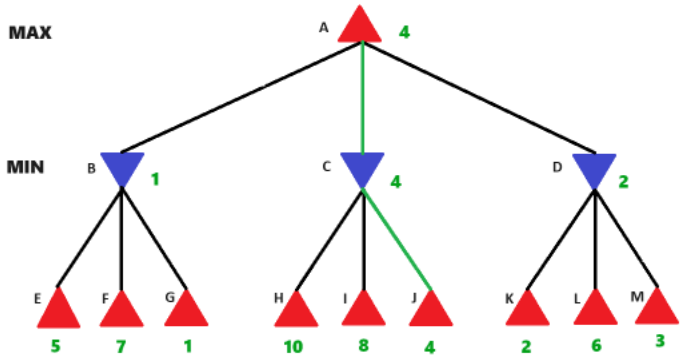


Figure 3.1 Minimax tree representation

The algorithm traverses using Depth First Search (DFS), starting from the head node to the leaf nodes. Upon reaching the leaf node, the utility function calculates the utility value for all nodes. Depending on the layer feature to which the node is connected, it selects the node with the maximum value for the maximizer and the node with the minimum value for the minimizer and moves it to the next layer. A similar calculation is performed on the upper layers to obtain the final value.

Algorithm 1 employs pseudo-code to demonstrate one possible implementation of the minimax algorithm.

Algorithm 1 Minimax pseudo-code

```
1:  function MINIMAX(node, isMax)
2:      if node is leaf then
3:          return the value of node
4:      if isMax then
5:          maxVal ← −∞
6:          for child of node do
7:              val ← MINIMAX(child, False)
8:              maxVal ← max(maxVal, val)
            return maxVal
9:      else
10:         minVal ← +∞
11:         for child of node do
12:             val ← MINIMAX(child, True)
13:             minVal ← min(minVal, val)
            return minVal
```

Despite the effectiveness of the Minimax Algorithm in simpler games like tic-tac-toe, its computational complexity becomes challenging for more complex games such as

chess due to the high branching factor of possible moves. Shannon [9] estimated that there are approximately 10^{120} possible games in chess. This estimation takes into account various factors, such as the average branching factor, average game length, and the number of legal positions in chess. Also, Arthur Samuel [24] estimated that there are approximately 10^{40} possible paths through a game of checkers. To address this issue, techniques like Alpha-Beta Pruning are employed to reduce the search space by eliminating certain branches without affecting the final result significantly [25].

3.2 Alpha–Beta Pruning

Alpha-Beta Pruning isn't a standalone algorithm but rather an enhancement applied to the Minimax algorithm commonly used in game theory. It significantly reduces calculation time by eliminating branches that are unlikely to affect the final decision [21]. It uses a prune technique that proves that there is no need to traverse certain parts of the tree expanded by the Minimax algorithm [26]. It allows for faster searching through the game tree and reaching lower levels more quickly. This is because it eliminates unpromising branches, enabling a more thorough exploration within the given timeframe.

This approach offers two additional parameters, alpha and beta, in addition to the Minimax algorithm, and these parameters form the name of the algorithm. Alpha represents the lowest score ensured for the maximizing player, whereas beta represents the highest score guaranteed for the minimizing player. At the start of the search process, the variables alpha and beta are initialized with values representing the worst possible outcomes for the players involved. Alpha is set to negative infinity, reflecting the worst score for the maximizing player, while beta is set to positive infinity, representing the worst score for the minimizing player. This initialization strategy ensures that the search algorithm commences with conservative estimates, enabling it to iteratively improve its evaluation as it navigates through the game tree.

As the algorithm traverses the search tree using a DFS strategy, the alpha and beta parameters are updated accordingly. Alpha is updated in the maximizer, while beta

is updated in the minimizer. If beta at the node is less than or equal to alpha, this indicates that the current node's subtree can be pruned because this node will never be chosen and a better alternative has already been found. This optimization technique significantly reduces computation time by eliminating the need to search in specific subtrees and allowing a faster and deeper search in the game tree [21, 27].

Figure 3.2 describes the Alpha-Beta Pruning variation of the game tree used in Figure 3.1. Branches with double red lines represent those that have been pruned.It is observed that the value and path calculated by Minimax and Alpha-Beta Pruning are the same.

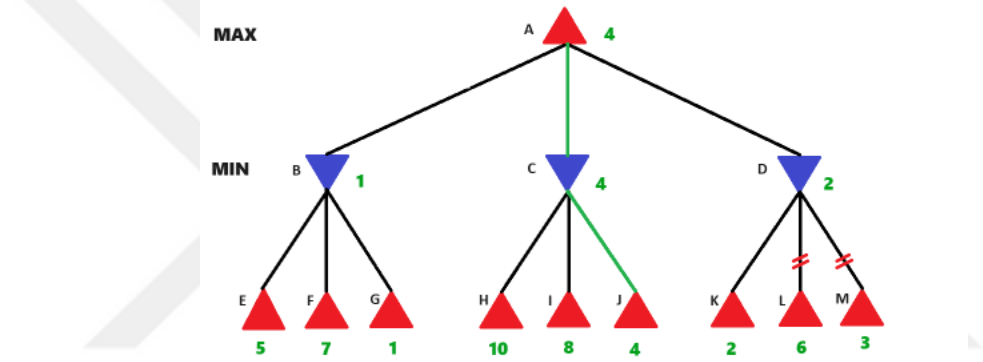


Figure 3.2 Alpha–Beta Pruning tree representation

Algorithm 2 employs pseudo-code to demonstrate one possible implementation of the Alpha-Beta Pruning algorithm. When the Alpha-Beta Pruning algorithm (Algorithm 2) is compared with the Minimax algorithm (Algorithm 1), two new parameters and pruning expressions are seen. Alpha and beta parameters are calculated in the child node and sent to its child nodes. With these values, the highest and lowest discovered values can be determined to maximize and minimize the node. When the pruning condition is met, it is ensured that other branches of the tree do not need to be explored. Pruning occurs when the current alpha value is greater than or equal to the beta value. This condition enables the algorithm to cut off branches in the game tree that do not require further exploration because a superior solution has already been found along a different path. By meeting this condition, the algorithm effectively minimizes the number of nodes requiring evaluation, which helps improve the effectiveness of the search procedure.

Algorithm 2 Alpha–Beta Pruning pseudo-code

```
1:  function MINIMAX(node, alpha, beta, isMax)
2:      if node is leaf then
3:          return the value of node
4:      if isMax then
5:          maxVal ← −∞
6:          for child of node do
7:              val ← MINIMAX(child, alpha, beta, False)
8:              maxVal ← max(maxVal, val)
9:              alpha ← maxVal
10:             if beta ≤alpha then
11:                 break
            return maxVal
12:     else
13:         minVal ← +∞
14:         for child of node do
15:             val ← MINIMAX(child, alpha, beta, True)
16:             minVal ← min(minVal, val)
17:             beta ← minVal
18:             if beta ≤alpha then
19:                 break
            return minVal
```

The efficiency of Alpha-Beta Pruning relies significantly on the sequence in which nodes are inspected. The order in which moves are considered is a crucial factor in Alpha-Beta Pruning. In situations where no branches are excluded, the algorithm behaves similarly to the Minimax algorithm in its worst-case scenario [21, 27].

3.3 Expectimax

Expectimax is an enhanced version of the Minimax algorithm developed for solving stochastic problems that have at least one probabilistic element.For example, the outcome of an action may lead to multiple states, each with a certain probability. It is used to solve games such as Backgammon and Pacman. In terms of design, it replaces the min layer with a chance layer. The chance layer calculates the weighted sum of the child nodes [28]. In the context of decision-making and game-playing, there are three distinct types of non-leaf nodes: max, min, and chance. The value of a node is defined as follows:

- Max node = The decision-making player tries to maximize utility.
- Min node = Opponent tries to minimize utility.
- Chance node = Represents uncertainty in the game.

These node types are used to represent the different strategies and outcomes in a game or decision-making process. Max nodes represent the best possible outcome, min nodes represent the worst possible outcome, and chance nodes represent the average outcome considering the likelihoods associated with each child node.

A sample expectimax tree is shown in Figure 3.3.

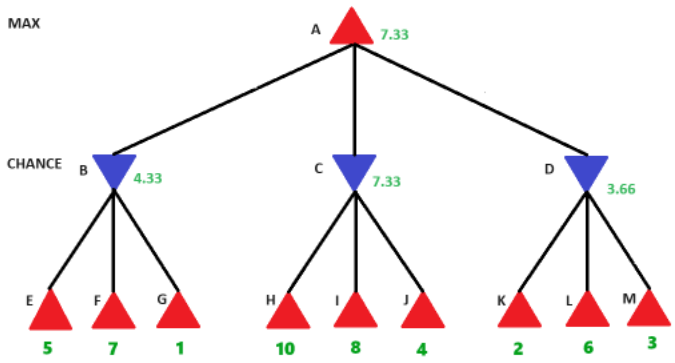


Figure 3.3 Expectimax tree representation

The computational complexity of Minimax, Alpha-Beta Pruning, and Expectimax algorithms is analyzed with regard to their worst-case time complexity. Minimax has a time complexity of $O(b^d)$, where b is the branching factor and d is the depth of the game tree. Alpha-Beta Pruning optimizes Minimax, potentially reducing its time complexity to $O(b^{d/2})$ in the best case while maintaining a worst-case complexity of $O(b^d)$. This improvement is achieved by pruning unnecessary branches during the search. Expectimax, designed for games with uncertainty or probability, has a time complexity of $O(k^d)$, where k is the average number of possible moves or chance events at each level and d is the depth of the tree. These complexities provide insights into how these algorithms scale with the size of the problem, but actual performance can vary based on factors like implementation details and the characteristics of the specific game being addressed.

Minimax-based algorithms pose a significant computational burden. The branching factor in a game denotes the average count of potential moves at a specific game state. This metric is of great importance in AI research focused on board games. It quantifies the number of children at each node within a game tree, demonstrating the average number of new states accessible from a particular state. For example, in the game of chess, the branching factor is about 35, while the game of go has an average branching factor of 250 [29]. Analyzing this metric is crucial for understanding the complexity of games and guiding the development of AI algorithms designed for game play. The branching factor in games like Go has shown that techniques such as Minimax, Alpha-Beta Pruning, and Expectimax are not sufficient [30].

3.4 Monte Carlo Tree Search

Monte Carlo sampling is a commonly used approach for estimating the solution to a problem by performing random trials. It is based on probability and statistical theories and is particularly useful for complex or analytically challenging problems. Initially applied for integral computations, Monte Carlo methods have become popular in physics and computer science owing to their adaptability and efficiency [4]. Monte-Carlo is first applied to games including chess, tic-tac-toe, and othello [31, 32].

Monte Carlo simulation-based search algorithms have demonstrated success in a variety of non-deterministic and imperfect information games over time. These algorithms have been applied in games like Scrabble [33], Poker [34], and Go [35], showcasing their effectiveness in handling complex decision-making processes in games with uncertainty and hidden information. Monte Carlo methods, particularly MCTS, have significantly advanced the highest development in game-playing AI, enabling computers to compete at high levels in strategic games that involve uncertainty and incomplete information.

In 2006, Coulom [36] introduced the MCTS framework, which integrated Monte Carlo search with an incremental tree structure. Kocsis and Szepesvári [37] developed the Upper Confidence Tree (UCT) algorithm, which is a tree search method based on the Upper Confidence Bounds (UCB) selection policy. UCT is the first and widely adopted

MCTS method. The algorithm utilizes the UCB1 formula derived from Auer et al.'s work [38].

MCTS is an approach used to identify the best decisions within a particular domain. It achieves this by randomly sampling the decision space and then constructing a search tree that reflects the insights gained from these exploratory simulations. It is a heuristic search algorithm that is especially suitable for decision-making and finding the next action in board games. The mastery of AlphaGo, an artificial intelligence program developed by Google's DeepMind team in 2016, which combines deep learning and MCTS, in the game of Go was a success previously unseen in artificial intelligence, sparking significant excitement within the AI community [2]. This approach is considered a significant advancement not only in the realm of Go but also in the broader field of AI. MCTS has been extensively applied across various combinatorial games and has even found application in certain real-time video games [39].The versatility of MCTS is evident in its numerous applications across various real-world problems, as highlighted by James (2017) [40].

MCTS is a hybrid approach that combines Monte Carlo tactics, which include random sampling and statistical analysis, with tree-based search methods. Contrary to traditional search algorithms that thoroughly examine the whole search universe, MCTS prioritizes sampling and investigating just high-potential areas within the exploration space [23].

The primary principle behind MCTS is the gradual building of a search tree by the execution of several randomized simulations, known as rollouts, originating from the present game state. The simulations persist until either a terminal condition or a certain depth is attained. The results of these simulations are then propagated back up the search tree, updating the data associated with the visited nodes, including the number of visits and the ratios of wins.

During the search process, MCTS efficiently maintains an equilibrium between exploring novel options and using existing knowledge. The system selects movements based on a combination of exploiting highly promising plays with high win

percentages and exploring less studied or undiscovered moves. To maintain equilibrium, a formula called the Upper Confidence Bound (UCB), such as UCT, is utilized to identify which movements or nodes should be investigated throughout the search process.

Exploration is essential for uncovering unexplored areas of the tree and potentially finding better paths. It prioritizes expanding the breadth of the search tree rather than focusing on deepening individual branches. However, exploration can become suboptimal in conditions with numerous stages or repeats. To tackle this issue, exploitation becomes a key factor in the decision-making process. Exploitation focuses on the most promising path with the highest estimated value, taking a greedy approach that extends the depth of the tree. In summary, the UCB formula employed by trees effectively manages the balance between exploration and exploitation by occasionally exploring less-traveled nodes. This process helps uncover potentially better paths than the ones currently being exploited.

In general, each node holds X_i (the current state value), N_i (visits the parent), and n_i (visits the node). The steps of the algorithm make sense of this data.

MCTS involves four main stages: selection, expansion, simulation, and backpropagation.

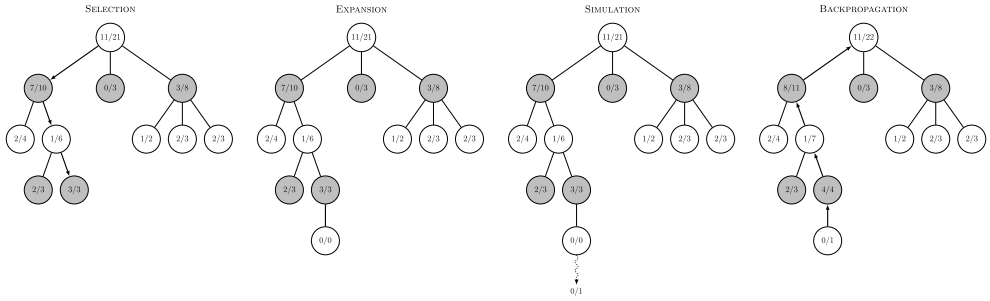


Figure 3.4 Monte Carlo Tree Search steps

3.4.1 Selection

Algorithm 3 MCTS Selection phase pseudo-code

```
1: function SELECT(n)
2:     while n is fully expanded and non-terminal do
3:         if n is not fully expanded then
4:             EXPAND(n)
5:             return a child node of n
6:         else
7:             n ← child node of n according to selection strategy
8:     return n
```

MCTS applies a search strategy that initiates at the head of the search tree and explores the tree from there. SELECT achieves an equilibrium between exploration and exploitation in the decision-making process. The selection stage is crafted to guarantee that the algorithm explores every node in the search tree at least once. The selection strategy calculates the value of each child node, and then the one with the highest value is selected as the new node. This process persists until the leaf node is encountered.

There are two different types of selection strategies: deterministic and stochastic. Since we are dealing with combinatorial games, deterministic selection policies that fall under the category of index-based multi-armed bandit policies are considered.

3.4.1.1 Selection Strategy

The purpose of the selection strategy is to strike a balance between exploring actions that haven't been completely tested and exploiting the actions that have shown the best results thus far [41].

Here we use the following notations:

- X_i = mean node reward
- n_i = visits of node
- N = visits parent

- C = constant [0, 1] to maintain an equilibrium between exploitation and exploration.

Now we introduce the deterministic selection strategies as follows:

- **UCT**: In 2006, Kocsis and Szepesvari [37] introduced the UCT strategy. The UCT strategy is an adaptation of the UCB method, which was originally developed for the multi-armed bandit problem [38] and is widely recognized as the most popular index-based policy for the multi-armed bandit problem.

$$UCB1(n_i) = X_i + C \times \sqrt{\frac{\ln N}{n_i}}$$

- **UCB1-Tuned**: It takes into account the variance of the empirical value of each arm, and outperforms UCT in all of their experiments.

$$V_j(s) = (\sum_{\gamma=1}^{s} X_{j,\gamma}^2) - \overline{X_{j,\gamma}^2} + \sqrt{\frac{2\log n}{s}}$$

$$UCB1Tuned(n_i) = \sqrt{\frac{\log N}{n_i} \times \min(1/4, V(n_i))}$$

where $V(n)$ is an upper confidence bound on the variance of the bandit.

3.4.2 Expansion

Algorithm 4 MCTS Expansion phase pseudo-code

```
1: function EXPAND(n)
2:     a ← Avaliable action of n
3:     while a is not null do
4:         Create childen node applying a to n
```

EXPAND involves creating one or more child nodes from a selected node by choosing an action and expanding the tree to explore potential future states. This step enables the algorithm to examine further potential moves or actions available in the current state. As the tree expands, new child nodes are created to represent these potential paths, thus allowing MCTS to explore additional outcomes and enhance its decision-making capabilities by delving deeper into the search space.

3.4.3 Simulation

Algorithm 5 MCTS Simulation phase pseudo-code

```
1: function SIMULATE(n)
2:     s ← State of node n
3:     while s is non-terminal do
4:         Choose action randomly
5:         Apply acion to get new state s′
6:         s ← s′
7:     Calculate reward from final state s
8:     return reward
```

SIMULATE progresses by making random moves, starting from the position of the newly created node for each player, until reaching the final state in the game [42].

3.4.4 Backpropagation

Algorithm 6 MCTS Backpropagation phase pseudo-code

```
1: function BACKPROPAGATE(n, reward)
2:     while n is not null do
3:         Update visit count and total reward of n
4:         Apply backpropagation strategy to n
5:         n ← Parent of n
```

Once the game concludes, BACKPROPAGATE backtracks to all the nodes follow the path starting from the last visited node in the tree up to the head. These nodes' values (S_i, n_i) are updated accordingly.

3.4.4.1 Backpropagation Strategy

A backpropagation strategy is utilized to compute the value of $node_i$. The most widely used and effective approach is the average strategy, which involves taking the mean of the outcomes from all simulated games conducted through that particular node [36].

Different backpropagation strategies have been suggested in the literature. Here, we present the backpropagation strategies as follows:

- **Average**: It computes the value by averaging the outcomes from all simulated games conducted through this node [43].
- **Max**: It backpropagates by choosing the maximum value among the child nodes [44].
- **Informed Average**: This strategy is designed to converge more quickly to the value of the best move compared to the *Average* strategy by designating a higher weight to the best moves [44].
- **Mix**: This strategy incorporates a weighting factor into the mean value of a node.

Final Move Selection

The algorithm's primary goal is to select the optimal move after completing all the steps in a given iteration. Here, we use the following methods:

- **Max child**: Choose the child with the greatest value.
- **Robust child**: Choose the child with the most number of visits.
- **Robust-max child**: Choose the child who has the highest number of visits and the highest value. If a child with the maximum robustness is not currently accessible, more simulations are performed until a kid with the maximum robustness is acquired, as recommended by Coulom [36].
- **Secure child**: Choose the child that maximizes a lower confidence bound.

CHAPTER 4

METHODOLOGY

4.1 Proposed Hybrid Approach

MCTS extracts statistics with the help of a game tree by playing random games to determine the best move. This data is derived from the outcomes of the games played, focusing on the information related to game victories. In terms of the algorithm's effectiveness, the count of iterations is considered an important parameter because it affects the number of games played. These iterations are based on repeating the four stages of the algorithm (selection, expansion, simulation, and backpropagation) a specific number of times. Increasing the number of iterations in MCTS can enhance its performance up to a certain point, known as the overfitting point, but it also leads to increased memory and CPU time consumption based on the branching factor in games.

In game theory, a utility function represents a player's preferences or objectives and depicts their situation at any given point in the game. It also indicates what players value and how they will behave when determining their strategies. For example, it can show the points a player would gain or lose as a result of a move. Different types of games may require different types of utility functions. For instance, in zero-sum games like chess, where one player's success comes at the expense of the other player, utility functions are shaped accordingly. In cooperative games, utility functions that consider the outcomes achieved by players as a group can be used.

This thesis introduces a novel MCTS algorithm based on UCB1 with weak utility. Creating a weak utility function can be done easily and quickly. Therefore, the easily developed weak utility function pre-feeds UCB1 and enhances its effectiveness. It is known that UCB1 in MCTS consists of two main components: exploitation and exploration [4]. In our algorithm, the exploitation term is redesigned to utilize the utility function, and the new approach is called UCB1 with weak utility, namely $UCB1_{WU}$. While the known UCB1 is used in MCTS, when the number of randomly played games is low, sufficient statistics for UCB1 are not generated. Therefore, a

healthy decision-making process cannot be realized for the next stages. However, the proposed approach's utility function allows for healthier decision-making with fewer statistics. This is the most important advantage of the proposed $UCB1_{WU}$. It is inspired by the Simulated Annealing algorithm in defining this weak utility function. As a result, in cases where there are a few iterations, the resulting information is supported by the utility function. Similarly, in cases where there are many iterations, statistical information becomes more important than utility.

4.2 Mathematical and Algorithmic Details

$UCB1$, employed during the selection phase of the MCTS algorithm, is defined as follows:

$$UCB1(node_i) = X_i + C \times \sqrt{\frac{\ln N}{n_i}}.$$

In the MCTS with Weak Utility algorithm, $UCB1_{WU}$ is proposed instead of UCB1, as follows:

$$UCB1_{WU}(node_i) = (\alpha \times U_i) + (1-\alpha) \times X_i + C \times \sqrt{\frac{\ln N}{n_i}},$$

where

$$\alpha = e^{-0.1 \times n_i}$$

and U_i is a utility function that is defined depending on the game.

Algorithm 7 shows the pseudocode of all phases of the MCTS algorithm and its working principle.

Algorithm 7 MCTS with Weak Utility pseudo-code

```
1: function MCTS(s_0)
2:     Create root node n_0 with state s_0
3:     while time allows do
4:         n_l ← SELECT(n_0)
5:         reward ← SIMULATE(n_l)
6:         BACKPROPAGATE(n_l, reward)
7:     return Best child of n_0
8: function SELECT(n)
9:     while n is fully expanded and non-terminal do
10:        if n is not fully expanded then
11:            EXPAND(n)
12:            return a child node of n
13:        else
14:            n ← Child node of n with max UCB1_WU
15:    return n
16: function EXPAND(n)
17:    a ← Avaliable action of n
18:    while a is not null do
19:        Create childen node applying a to n
20: function SIMULATE(n)
21:    s ← State of node n
22:    while s is non-terminal do
23:        Choose action randomly
24:        Apply action to get new state s'
25:        s ← s'
26:    Calculate reward from final state s
27:    return reward
28: function BACKPROPAGATE(n, reward)
29:    while n is not null do
30:        Update visit count and total reward of n
31:        Apply backpropagation strategy to n
32:        n ← Parent of n
```

4.3 Complexity Analysis

The time complexity of the MCTS algorithm is often measured by the number of iterations it conducts. During each iteration, MCTS constructs and explores a tree layout, testing various actions and assessing their potential consequences. This complexity can fluctuate based on elements like the game tree's branching factor and the depth of the search.

In general, the theoretical time complexity of MCTS is often described as $O(CN)$ where:

- C : Computational cost per iteration encompasses the combined expense of selecting, simulating, and backpropagating. This involves evaluating a state, running simulations for actions, and adjusting the tree structure.

  The complexity of the selection phase is typically estimated as $O(bd)$, where b represents the branching factor and d denotes the depth of the tree. This estimate arises from the fact that during selection, the algorithm traverses through each child node at every level of the tree.

  The simulation phase generally depends on the depth of the tree, but the actual running time can vary significantly depending on how the algorithm is implemented.

  The backpropagation phase also takes time proportional to the depth of the tree, so it has a complexity of $O(d)$.

- N : Represents the total number of iterations performed by the algorithm.

Therefore, the total time complexity of MCTS is $O((bd + simulation + d)N)$. This analysis outlines the general scalability of MCTS through its time complexity. However, real-world performance depends on various factors, like the game tree's branching factor, the complexity of the game's state space, and how the algorithm is implemented.

CHAPTER 5

RESULTS AND ANALYSIS

5.1 Test Games

This section thoroughly investigates the rules, scoring mechanisms, and utility calculations of the games, comparing the algorithms mentioned in the literature with the proposed algorithm. Algorithm 8 specifies the generic functions utilized across all games to compute the utility value. The utility value for a player is derived by inputting their scores into the GetUtility function.

Algorithm 8 Generic functions for calculating utility

```
1:  function Normalize(s1, s2)
2:      return (s1 − s2)/(s1 + s2 + ε)
3:  function GetUtility(game, s1, s2)
4:      scoreMap ← empty map
5:      if game is non-terminal then
6:          Add Normalize(s1,s2) to scoreMap for player
7:          Add Normalize(s2,s1) to scoreMap for opponent
8:      else
9:          if s1 > s2 then
10:             Add 1.0 to scoreMap for player
11:             Add 0.0 to scoreMap for opponent
12:         else if s1 < s2 then
13:             Add 0.0 to scoreMap for player
14:             Add 1.0 to scoreMap for opponent
15:         else
16:             Add 0.5 to scoreMap for player
17:             Add 0.5 to scoreMap for opponent
18:     return score
```

In Algorithm 8, the tolerance parameter is $\varepsilon = 10^{-5}$.

5.1.1 Tic-Tac-Toe

An ancient game with roots dating back to ancient Egypt, around 1300 BCE, is known today as Naughts and Crosses or Tic-tac-toe and has maintained its popularity

throughout history [45]. During the Middle Ages, it was referred to by various secret societies as the "Cabala of the Nine Chambers." During this period, people believed that placing numbers from 1 to 9 on a grid in a way that their sums would be the same horizontally, vertically, or diagonally would convey a numerological message about the world. Over time, the game transformed and is now a simple game played on a three-by-three grid, where players alternately place **X** and **O** symbols, aiming to achieve a horizontal, vertical, or diagonal line of their symbols to win [46].

Figure 5.1 5*x*5 Tic-Tac-Toe board

In our testing environment, we've devised a game where players take turns placing either an **X** or **O** symbol on an $N \times N$ grid, as depicted in Figure 5.1. The game proceeds until there are no empty spaces remaining on the grid. When the game concludes, the score is determined by summing the lengths of all series containing three or more identical symbols found in rows, columns, or diagonals.

Algorithm 9 demonstrates how the score is calculated.

Algorithm 9 Score Function for Tic-Tac-Toe

```
1: function GetScore(board)
2:     player ← the sum of the series that greater than 3
3:     opponent ← the sum of the series that greater than 3      ▷ Horizontal, vertical
   and diagonal series are calculated
4:     Add player to scoreMap for player
5:     Add opponent to scoreMap for opponent
6:     return score
```

5.1.2 Mangala

"Mangala," also known as Mancala in various regions, is a traditional Turkish game renowned for its demands on intelligence, calculation, and focus. It stands as one of the oldest games, embodying strategic depth and cultural significance [47].

The game takes place on a board with 12 pits and 2 treasure pits, totaling 48 stones. The board's layout, depicted in Figure 5.2, comprises six pits allocated to each player, forming distinct territories along with a treasure pit.

Figure 5.2 Mangala board

The game begins with 48 stones evenly distributed among 12 pits. Players aim to accumulate the most stones in their treasure pit by strategically moving stones from

other pits. It's not just about taking stones but also about anticipating and countering your opponent's moves. Successful strategies involve forward planning, maintaining control over the mid-game, and making strategic opening moves. Being cautious against potential captures and observing the opponent's moves are crucial for victory [48]. This game is a classic challenge that tests mathematical intelligence and strategic mastery, pushing players to defeat their opponents with more effective stone movements.

Algorithm 10 demonstrates how the score is calculated.

Algorithm 10 Score Function for Mangala

```
1: function GetScore(board)
2:     scoreMap ← empty map
3:     Add player treasure to scoreMap for player
4:     Add opponent treasure to scoreMap for opponent
5:     return score
```

5.1.3 Checkers

"Draughts," commonly known as checkers, is a widely enjoyed game boasting over 150 documented variations worldwide. These variations encompass diverse rules, board dimensions, and starting positions, reflecting the game's adaptability to various cultural and regional preferences [49].

Checkers is a classic two-player game played on an 8x8 board with 64 squares, akin to a chessboard. Each player begins with 12 pieces, positioned on the dark squares of the board as depicted in Figure 5.3. Initially, these pieces are "normal" and can only move forward. However, when a piece advances to the opponent's first row, it ascends to the status of a "king," granting it the ability to move both forward and backward.

In Checkers, players have two primary actions at their disposal: "move" and "jump." A "move" consists of shifting a piece to an adjacent square, while a "jump" occurs when the opponent's piece is diagonally adjacent to an empty square immediately behind the player's piece. Executing a jump results in capturing the adjacent piece, removing it from the board [50].

Mastering checkers demands strategic planning and foresight to outmaneuver the opponent. Success hinges on adeptly employing moving and jumping tactics, making every move count in this timeless board game.

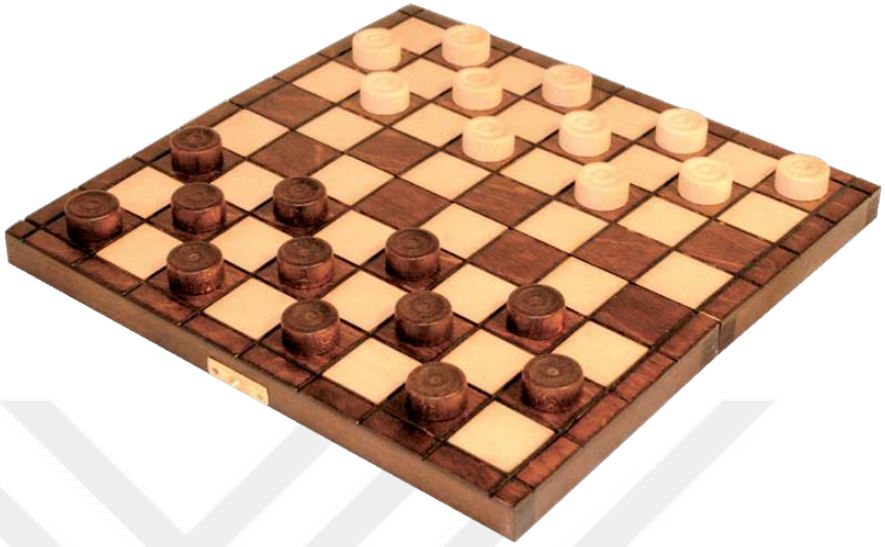


Figure 5.3 Checkers board

There are two ways for a player to win: either by capturing all of the opponent's pieces or by putting the opponent in a position where they cannot make any moves. If neither side achieves these conditions, the game ends in a draw.

Algorithm 11 demonstrates how the score is calculated.

Algorithm 11 Score Function for Checkers

```
1: function GetScore(board)
2:     scoreMap ← empty map
3:     blackScore ← blackNonKingCount + blackKingCount × 1.5
4:     whiteScore ← whiteNonKingCount + whiteKingCount × 1.5
5:     Add blackScore to scoreMap for black player
6:     Add whiteScore to scoreMap for white player
7:     return score
```

5.2 Test Environment

5.2.1 Platform and Tools

Various tools, programming languages, hardware, and operating systems were utilized throughout this thesis to facilitate the development process.

- **Programming Language**: Java
- **JDK**: 17.0.7
- **IDE**: IntelliJ IDEA 2023.1.2
- **Libraries**: JavaFx 11+, Aspose
- **Processor**: Intel(R) Core(TM) i5-8300H CPU (2.30 GHz or 4.0 GHz in turbo)
- **Memory**: 16.0 RAM
- **Operating System**: Windows 11 Education

Java is an object-oriented programming language known for its high-level features and unique syntax. The Java SE platform encompasses a comprehensive suite of components that include a virtual machine, development tools, deployment technologies, and various class libraries and toolkits commonly used in Java applications. This platform facilitates the creation of robust and scalable software solutions across a variety of computing environments.

5.2.2 Class Diagrams

The software design has been structured according to the game description outlined by Russell [5]. In Figure 5.4, the foundational classes constituting the overall software design are illustrated. Among these, the *Action* class is abstract, designed to execute actions pertinent to the game state. The abstract class *GameState* encapsulates the game rules and utility functions, designed as a generic class to accommodate various game

types. Class definitions for actions and states based on games are depicted in Figures 5.5 and 5.6, respectively.

The abstract class *IPlayer* defines a player within the context of the *GameState* and *Action*, guiding gameplay through selected algorithms. Figure 5.7 showcases *IPlayer* instances generated from the fusion of *ISearchAlgorithm* and *GameState* entities.

Furthermore, the *ISearchAlgorithm* abstract class serves as the foundation for all search algorithms employed within the software. Figure 5.8 offers an intricate depiction of the design underlying the search algorithms utilized in this study.

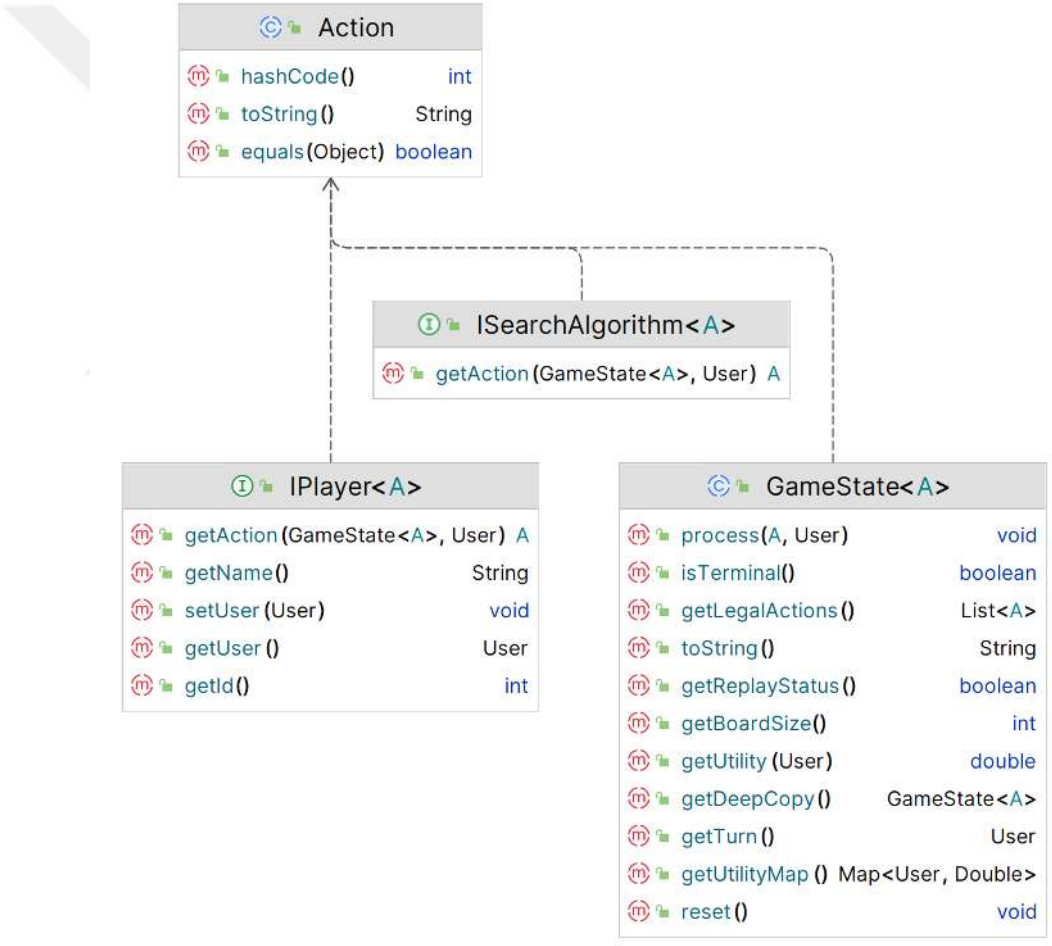


Figure 5.4 Defining standard game components

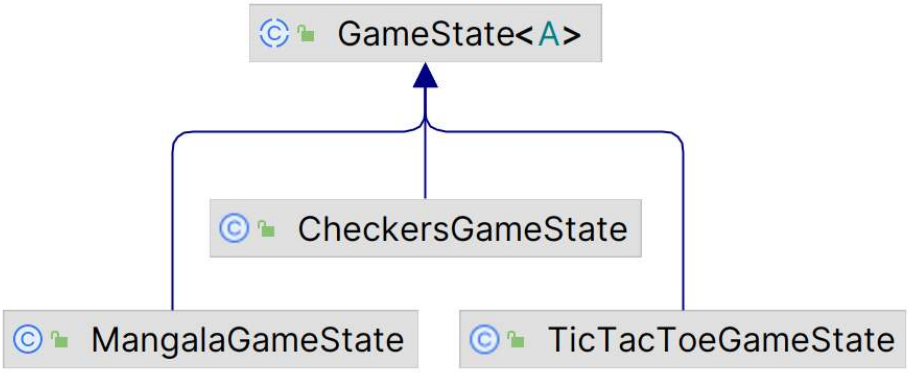


Figure 5.5 State relations of games

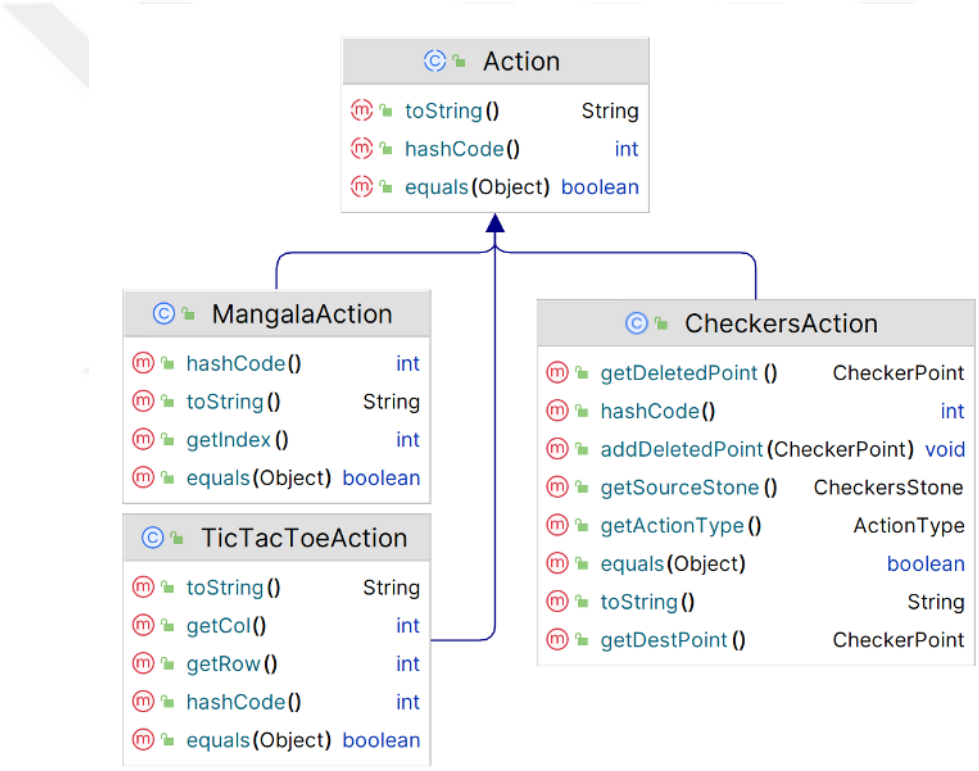


Figure 5.6 Action relations of games

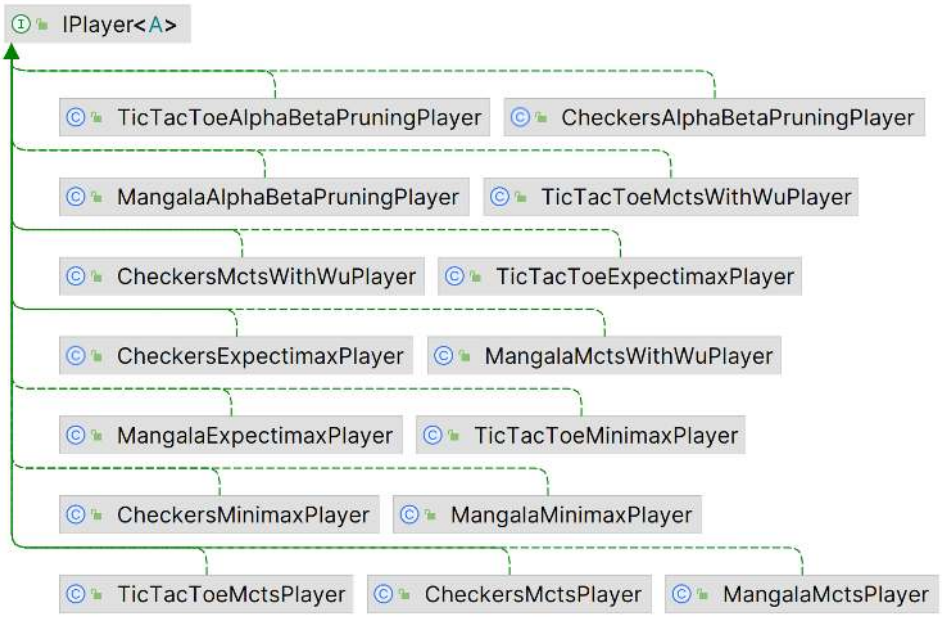


Figure 5.7 Player descriptions in games according to actions

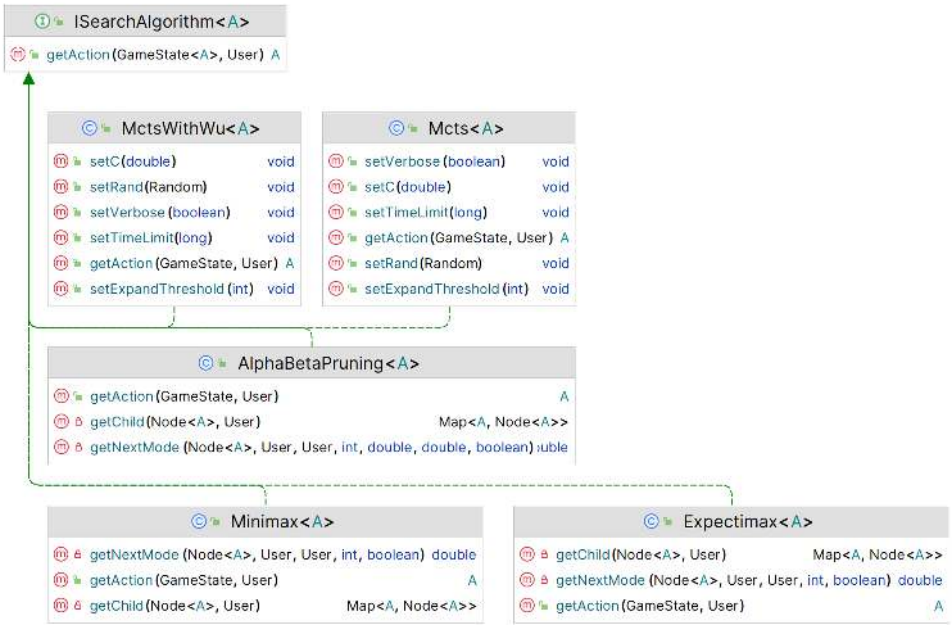
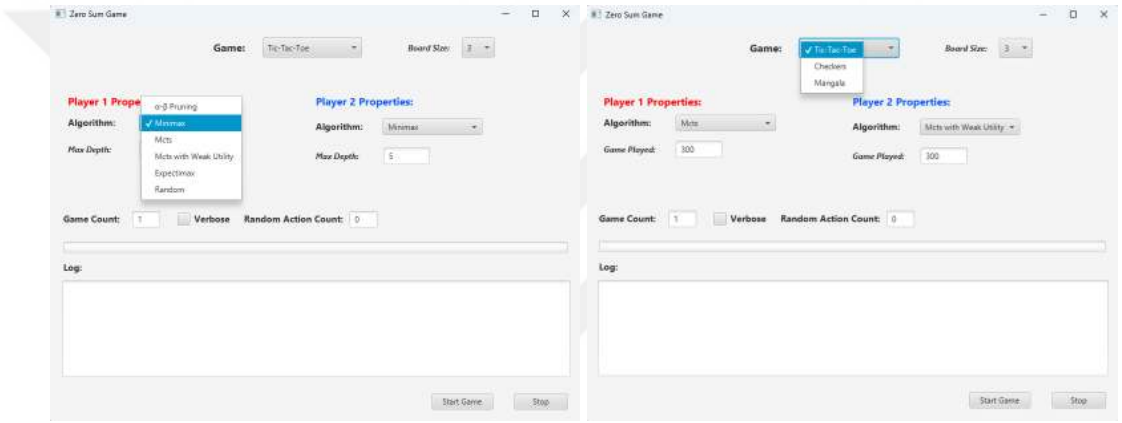


Figure 5.8 Definitions of search algorithms

We've developed an object-oriented design capable of hosting multiple games and various adversarial search algorithms. This approach streamlines our experimentation process and enables seamless integration of new games and methods. Java was selected for its robust support of object-oriented programming, streamlined code development, and widespread adoption.

5.3 Experimental Design

The diagram presented in Figure 5.9 illustrates our application's comprehensive structure, designed to facilitate the comparison of established algorithms in the literature with our proposed method across specific gaming scenarios. The architecture of the application is meticulously crafted to seamlessly integrate new algorithms and games. Currently, the application offers three test games: Mangala, English Checkers, and Tic-Tac-Toe. Within these gaming environments, a range of algorithms can be run, including Random, Minimax, Alpha-Beta Pruning, Expectimax, MCTS, and our suggested "Monte Carlo Tree Search with Weak Utility."



(a) Algorithm view (b) Game view

Figure 5.9 The general overview of the application

The success of Minimax-based algorithms depends on choosing the correct maximum depth, while for MCTS-based algorithms, selecting the appropriate number of iterations is crucial. The application has the choice of maximum depth for Minimax-based algorithms and the selection of the number of iterations for MCTS-based algorithms.

The number entered in the "Game Count" field determines the total number of games played, which is twice that number. In half of these games, Player-1 initiates the game, while in the remaining half, Player-2 takes the first turn. The cumulative outcome obtained up to these games is represented as a percentage in the log section. The "verbose" field is employed to monitor the game states and identify the actions taken by each player.

The specified number of random game plays is carried out with the "Random Action Count" field until the game reaches a certain stage. Following this, the selected algorithms take over playing the game from this stage onwards. With this option, algorithms can be tested not only from the initial stage but also from a certain stage that the game has reached.

5.4 Comparative Analysis with Baseline Algorithms

The study will compare the performances of the "Monte Carlo Tree Search with Weak Utility" algorithm with Alpha-Beta Pruning and MCTS algorithms in Tic-Tac-Toe, Mangala, and Checkers games. While "game played" will be used as a performance metric for Monte Carlo Tree Search with Weak Utility and MCTS algorithms, the "max depth" parameter will be employed for the Alpha-Beta Pruning algorithm. This comparison will be evaluated in terms of the effectiveness of the algorithms in each of the three types of games.

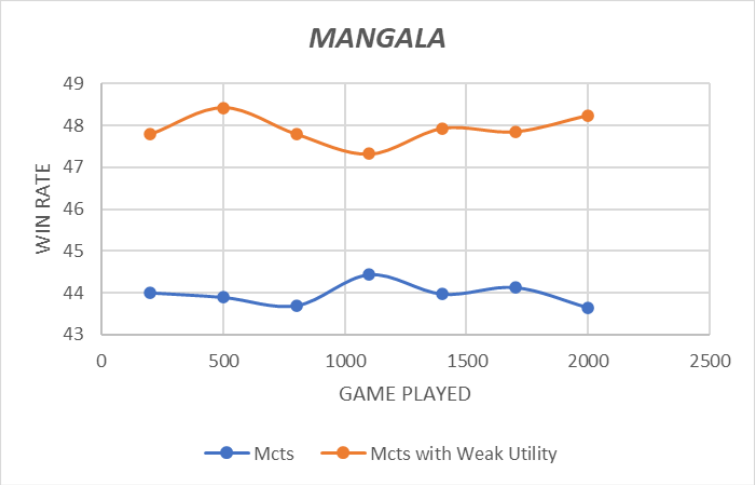
5.4.1 Game Play Performance

In this section, the superiority of the algorithms against each other is determined through board games. The algorithm with the higher winning percentage in the tested games is considered more successful than the others. Three tournaments have been defined for the evaluation process of the algorithms. Parameters are determined for each tournament according to the characteristics of the algorithms. In tournaments where MCTS-based algorithms are used, the "game played" parameter represents the count of iterations when making a decision. In tournaments where the Alpha-Beta Pruning algorithm is used, the "max depth" parameter represents the maximum game tree depth to be examined when making a decision. "Game count" refers to the sum of games played in the tournament. In these tournaments, the first player starts the game in half of the games, and the second player starts the game in the remaining half, so the one who starts the game does not have an advantage. Designed tournaments and their features are listed below:

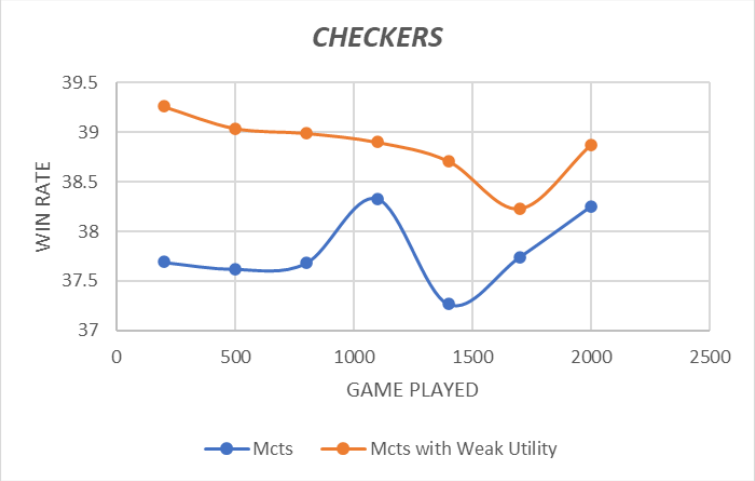
Tournament-1 Properties:

- **Player 1**: MCTS
- **Player 2**: MCTS with Weak Utility
- **Game Played**: 100, 500, 800, 1100, 1400, 1700, 2000
- **Game Count**: 10000
- **Test Game**: 5×5 Tic-Tac-Toe, Mangala, and Checkers

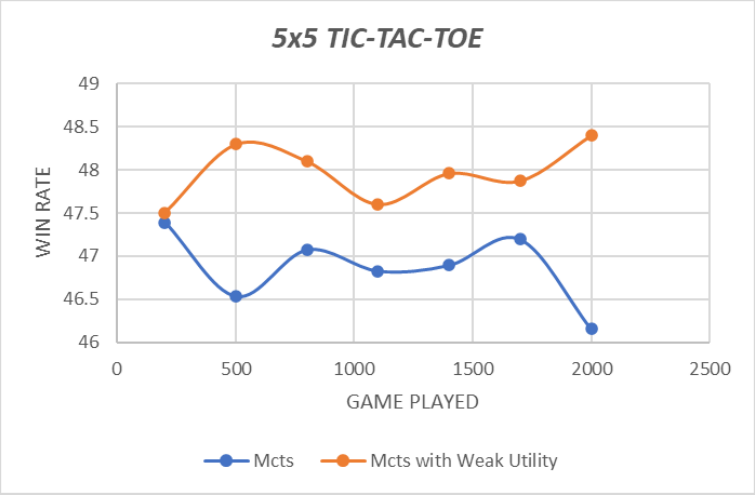
In Tournament-1 the performances of MCTS and MCTS with Weak Utility are compared numerically. The superiority of the algorithms is demonstrated in Figure 5.10. As seen in Figure 5.10, MCTS with Weak Utility has a higher win rate than MCTS in the same iteration.



(a) Mangala: Game played versus Win rate



(b) Checkers: Game played versus Win rate



(c) 5x5 Tic-Tac-Toe: Game played versus Win rate

Figure 5.10 Tournament-1 Game Play Performance

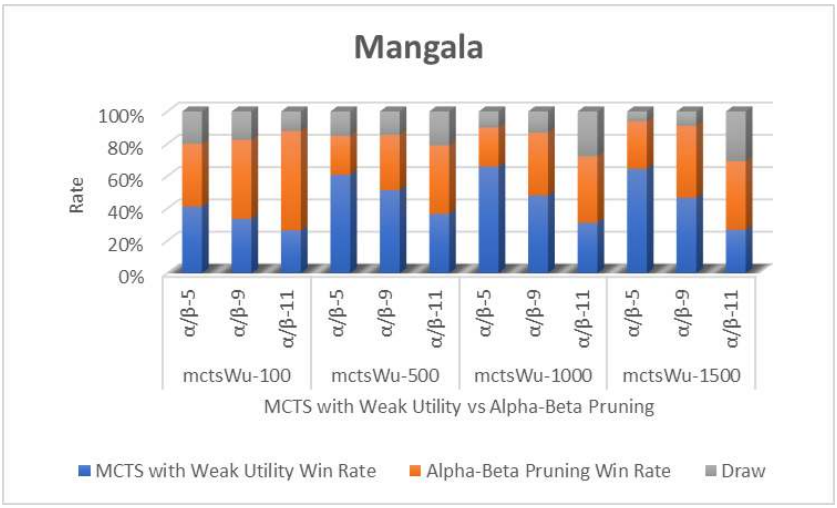
Tournament-2 Properties:

- **Player 1**: MCTS with Weak Utility
- **Player 2**: Alpha–Beta Pruning
- **Game Played**: 100, 500, 1000, 1500
- **Max Depth**: 4, 5, 6, 7, 8, 9 ,11
- **Game Count**: 2000
- **Test Game**: 5×5 Tic-Tac-Toe, Mangala, and Checkers

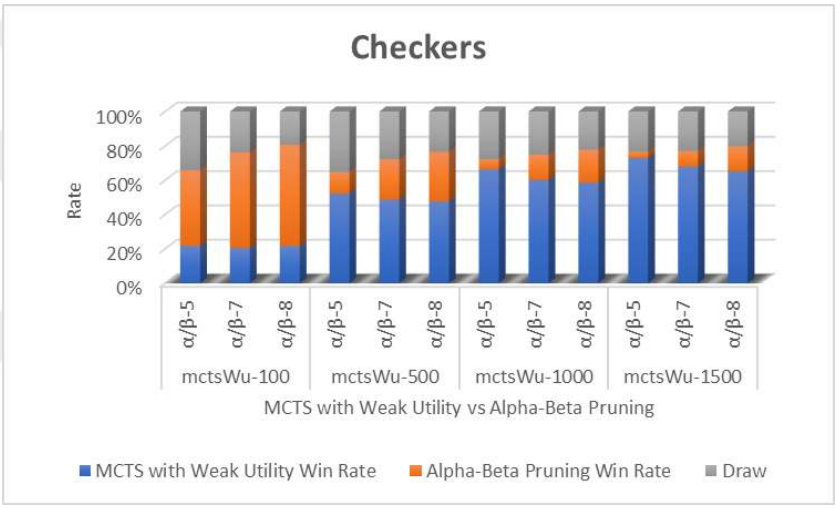
In Tournament-2 the performances of MCTS with Weak Utility and Alpha-Beta Pruning are compared numerically. The superiority of the algorithms is demonstrated in Figure 5.11. In Figure 5.11, α/β-*X* means the Alpha-Beta Pruning algorithm with max depth X. For example, α/β-*4* represents the Alpha-Beta Pruning algorithm with a maximum depth of 4. The maximum depth used in the Alpha-Beta Pruning algorithm varies from game to game and computer performance. In Figure 5.11 *mctsWu-X* means that MCTS with Weak Utility algorithm with game played X. For example, *mctsWu-200* represents MCTS with Weak Utility algorithm with a game played at 200.

In Figure 5.11, the red cells represent non-played games. Green cells represent games played. The winning rates are presented in green cells. For example, in Mangala, α/β-*5* plays against *mctsWu-100*, while the winning rate of α/β-*5* is 38.85%, the winning rate of *mctsWu-100* is 41.15%. So the winner is *mctsWu-100*.

As seen in Figure 5.11, with the increase of the game played parameter of MCTS with Weak Utility, the performance of MCTS with Weak Utility becomes better than the Alpha-Beta Pruning algorithm at different maximum depths.



(a) Mangala



(b) Checkers

(c) 5x5 Tic-Tac-Toe

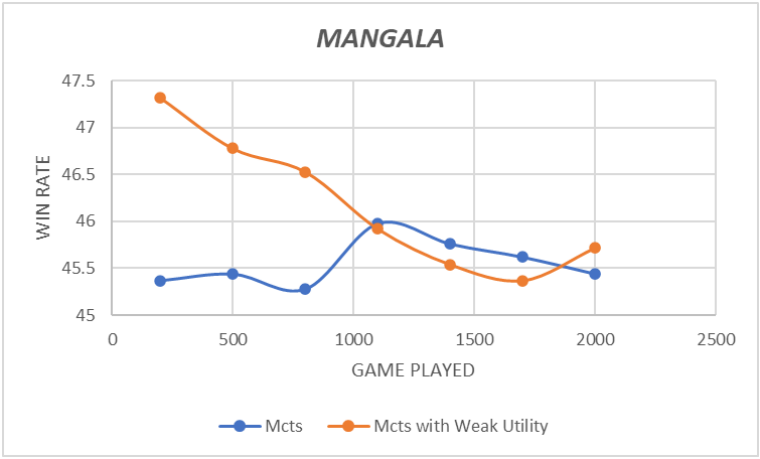
Figure 5.11 Tournament-2 Game Play Performance. Win rate of y-axis player against x-axis player.

Tournament-3 Properties:

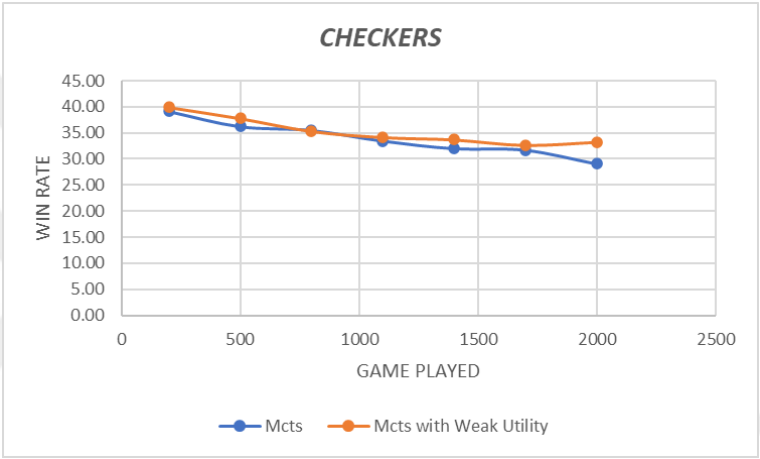
- **Player 1**: MCTS
- **Player 2**: MCTS with Weak Utility
- **Game Played**: 100, 500, 800, 1100, 1400, 1700, 2000
- **Number Of Random Action**: 5
- **Game Count**: 2000
- **Test Game**: 5×5 Tic-Tac-Toe, Mangala, and Checkers

Evaluating algorithms by starting at the beginning of the game isn't sufficient for testing different methods. Therefore, it's necessary to also assess their performance when the game starts from different board positions. To accomplish this, after a certain number of random moves are made from the beginning of the game, algorithms can be engaged to conduct this comprehensive testing.

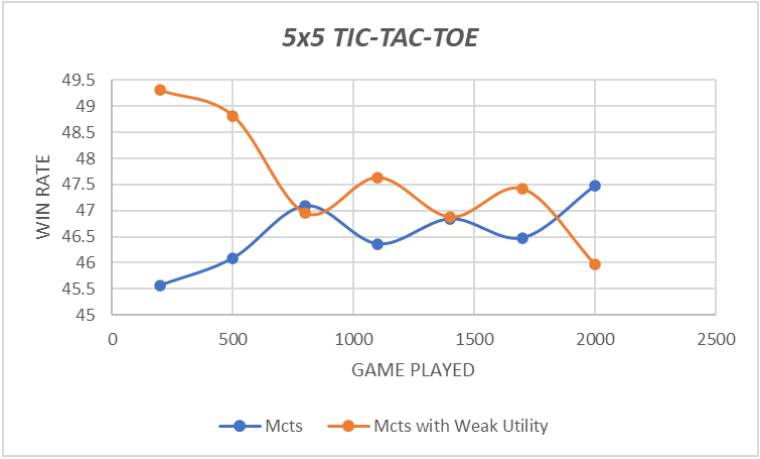
In this tournament, the players start by making 5 moves each randomly. Afterwards, the algorithms take over the game from this point onward. This is the main difference between this tournament and Tournament-1. The performances of the MCTS and MCTS with Weak Utility are compared for Tournament-3 and the ontained numerical results are illustrated in Figure 5.12.



(a) Mangala



(b) Checkers



(c) 5x5 Tic-Tac-Toe

Figure 5.12 Tournament-3 Game Play Performance

It can be observed that the superiority of MCTS with Weak Utility is decreased for Tournament-3. This is because the five random moves made before the algorithms were

activated in Tournament-3 reduced the effectiveness of the utility function. Moreover, we see that the weak utility function becomes meaningless as MCTS reaches the endgame with fewer moves as the game progresses.

5.4.2 Time Analysis

Table 5.1 shows the time in milliseconds required for the MCTS, MCTS with Weak Utility, and Alpha-Beta Pruning algorithms to initiate their first actions in games of Tic-Tac-Toe, Mangala, and Checkers. In Table 5.1, α/β-*X* means the Alpha-Beta Pruning algorithm with max depth X. For example, α/β-*4* represents the Alpha-Beta Pruning algorithm with a maximum depth of 4. The maximum depth used in the Alpha-Beta Pruning algorithm varies from game to game and computer performance. In Table 5.1 *Mcts-X* means that MCTS algorithm with game played X. For example, *Mcts-200* represents the MCTS algorithm with a game played at 200.

Table 5.1 Algorithms time analysis

	5x5 Tic-Tac Toe	Mangala	Checkers
Mcts-100	24	11	50
Mcts-500	61	21	171
Mcts-1000	95	35	263
MctsWu-100	41	10	42
MctsWu-500	150	30	185
MctsWu-1000	248	44	252
α/β-5	1274	14	21
α/β-9	-	647	241
α/β-11	-	4245	944

As seen in Table 5.1, the times for MCTS with Weak Utility are slightly longer than classical MCTS. Utility calculation in the selection phase of the MCTS with Weak utility algorithm slowed down the algorithm. The duration could differ based on how intricate the utility function is within the games. For instance, the time in Checkers is less than in Mangala and Tic-Tac-Toe. The times for Alpha-Beta Pruning increase as the search depth increases. The depths of 5, 9, and 11 given in Table 5.1 are selected for three levels that the computer can calculate. The maximum search depth of the Alpha-Beta Pruning algorithm mentioned in Section 5.2.1 is 5 for 5x5 Tic-Tac-Toe,

while it is 11 for Checkers and Mangala.

5.4.3 Memory Analysis

Minimax-based algorithms construct a tree of game states, where each node represents a state and edges represent possible actions. This tree is traversed by DFS. While Minimax and Expectimax traverse the entire tree to find the best strategy, Alpha-Beta Pruning may not need to explore the entire tree if pruning occurs. While reducing the time complexity, the space complexity is O(BD_m), where the branching factor is B and the maximum depth is D_m [51].

Analyzing the memory usage of MCTS involves assessing how it manages memory during the search process. MCTS creates a tree structure where each node depicts a possible game state and edges depict potential actions. The amount of memory needed to store this tree depends on factors like the game's branching factor and search depth. Each node typically holds information such as the game state, pointers to child and parent nodes, and visit counts. The memory complexity is primarily determined by the count of nodes and the data stored in each node.

CHAPTER 6

CONCLUSIONS

6.1 Summary of Key Findings

The aim of this study is to develop a new algorithm by incorporating a basic utility function into Monte Carlo Tree Search (MCTS). Additionally, games and a testing environment will be developed to compare the performance of the algorithms. The metrics obtained from the testing environment indicate that the MCTS with Weak Utility algorithm outperformed the standard MCTS algorithm in terms of game-winning percentage. However, the inclusion of the utility function has increased the time taken by the algorithm to make decisions.

6.2 Contributions to the Field

This study significantly contributes to effectively managing the balance between exploring new possibilities and exploiting existing knowledge in MCTS. The introduced utility function enhances the algorithm's ability to make informed decisions, enabling it to identify the best strategy with fewer simulations.

6.3 Recommendations for Future Research

In this study, we used Tic-Tac-Toe, Mangala, and English Checkers, all of which are described as perfect information and deterministic games, for testing purposes. We developed very basic utility functions for these games. Future studies could explore more advanced utility functions and investigate different types of games with higher branching factors. Additionally, as observed in Table 5.1, integrating utility into the MCTS algorithm increased the decision-making time. Techniques such as parallelization can be applied to the algorithm design to reduce calculation time.

REFERENCES

[1] Campbell M., Hoane A., and hsiung Hsu F., "Deep blue," *Artificial Intelligence*, vol. 134, no. 1, pp. 57–83, 2002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370201001291

[2] Chen J. X., "The evolution of computing: Alphago," *Computing in Science & Engineering*, vol. 18, no. 4, pp. 4–7, 2016.

[3] Perry A. w. b. B., "The surprisingly ancient history of board games." [Online]. Available: https://www.history.co.uk/articles/history-of-board-games

[4] Browne C. B., Powley E., Whitehouse D., Lucas S. M., Cowling P. I., Rohlfshagen P., Tavener S., Perez D., Samothrakis S., and Colton S., "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012.

[5] Russell S. and Norvig P., *Artificial Intelligence: A Modern Approach*, 3rd ed. Prentice Hall, 2010.

[6] Jurafsky D. and Martin J., *Speech and language processing (3rd ed. draft)*. Pearson, 2021.

[7] Szeliski R., *Computer vision: algorithms and applications*. Springer, 2022.

[8] Siciliano B. and Khatib O., Eds., *Springer handbook of robotics*. Springer, 2016.

[9] Shannon C. E., "Xxii. programming a computer for playing chess," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 41, no. 314, pp. 256–275, 1950.

[10] Shaker N., Togelius J., and Nelson M., *Procedural Content Generation in Games*. Springer, 01 2016.

[11] Sutton, R. S., . B., *Reinforcement learning: An introduction (2nd ed.)*. The MIT Press, 2018.

[12] Yannakakis G. and Togelius J., *Artificial Intelligence and Games*. Springer, 01 2018.

[13] Yannakakis G. N., Karpouzis K., Paiva A., and Hudlicka E., "Emotion in games," in *Affective Computing and Intelligent Interaction - Fourth International Conference, ACII 2011, Memphis, TN, USA, October 9-12, 2011, Proceedings, Part II*, ser. Lecture Notes in Computer Science, D'Mello S. K., Graesser A. C., Schuller B. W., and Martin J., Eds., vol. 6975. Springer, 2011, p. 497. [Online]. Available: https://doi.org/10.1007/978-3-642-24571-8_62

[14] Millington I. and Funge J., *Artificial Intelligence for Games*. Morgan Kaufmann, 01 2009.

[15] Li X., Meng M., Hong Y., and Chen J., "A survey of decision making in adversarial games," 2022.

[16] von Neumann J. and Morgenstern O., *Theory of games and economic behavior*. Princeton University Press, 1947.

[17] Singhal S. P. and Sridevi M., "Comparative study of performance of parallel alpha beta pruning for different architectures," in *2019 IEEE 9th International Conference on Advanced Computing (IACC)*. IEEE, 2019. [Online]. Available: http://dx.doi.org/10.1109/IACC48062.2019.8971591

[18] Zaky A., "Minimax and expectimax algorithm to solve 2048 (2014)."

[19] Dasgupta P. and Collins J. B., "A survey of game theoretic approaches for adversarial machine learning in cybersecurity tasks," *CoRR*, vol. abs/1912.02258, 2019. [Online]. Available: http://arxiv.org/abs/1912.02258

[20] Cleaver N. and Neshatian K., "Transfer learning in discrete zero-sum games," 2020.

[21] Jan 2023. [Online]. Available: https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/

[22] Melkó E. and Nagy B., "Optimal strategy in games with chance nodes." *Acta Cybern.*, vol. 18, pp. 171–192, 01 2007.

[23] "ML | Monte Carlo Tree Search (MCTS) - GeeksforGeeks — geeksforgeeks.org," https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts, [Accessed 30-May-2023].

[24] Samuel A. L., "Some studies in machine learning using the game of checkers," *IBM Journal of Research and Development*, vol. 3, no. 3, pp. 210–229, 1959.

[25] Knuth D. E. and Moore R. W., "An analysis of alpha-beta priming '," 2002. [Online]. Available: https://api.semanticscholar.org/CorpusID:7894372

[26] Fuller S., Gaschnig J., Gillogly J., of COMPUTER SCIENCE. C.-M. U. P. P. D., and Department C. M. U. C. S., *Analysis of the Alpha-beta Pruning Algorithm*. Carnegie-Mellon University. Department of Computer Science, 1973. [Online]. Available: https://books.google.com.tr/books?id=cOTmlwEACAAJ

[27] "Alpha-beta pruning," https://www.javatpoint.com/ai-alpha-beta-pruning, [Accessed 14-01-2024].

[28] Hauk T. and Buro M., "Search in trees with chance nodes," 03 2004.

[29] "Branching factor," https://en.wikipedia.org/wiki/Branching_factor, [Accessed 14-01-2024].

[30] Gelly S., Kocsis L., Schoenauer M., Sebag M., Silver D., Szepesvári C., and Teytaud O., "The grand challenge of computer go: Monte carlo tree search and extensions," *Communications of the ACM*, vol. 55, pp. 106–113, 03 2012.

[31] Abramson B., "Expected-outcome: a general model of static evaluation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, no. 2, pp. 182–193, 1990.

[32] Brügmann B., "Monte carlo go," Citeseer, Tech. Rep., 1993.

[33] Sheppard B., "World-championship-caliber scrabble," *Artificial Intelligence*, vol. 134, no. 1-2, pp. 241–275, 2002.

[34] Billings D., Davidson A., Schaeffer J., and Szafron D., "The challenge of poker," *Artificial Intelligence*, vol. 134, no. 1-2, pp. 201–240, 2002.

[35] Bouzy B. and Helmstetter B., "Monte-carlo go developments," in *Advances in computer games*. Springer, 2004, pp. 159–174.

[36] Coulom R., "Efficient selectivity and backup operators in monte-carlo tree search," in *International conference on computers and games*. Springer, 2006, pp. 72–83.

[37] Kocsis L., Szepesvári C., and Willemson J., "Improved monte-carlo search," *Univ. Tartu, Estonia, Tech. Rep*, vol. 1, 2006.

[38] Auer P., Cesa-Bianchi N., and Fischer P., "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2, pp. 235–256, 2002.

[39] de Waard M., Roijers D. M., and Bakkes S. C., "Monte carlo tree search with options for general video game playing," in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 2016, pp. 1–8.

[40] James S., Konidaris G., and Rosman B., "An analysis of monte carlo tree search," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, no. 1, Feb. 2017. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/11028

[41] Świechowski M., Godlewski K., Sawicki B., and Mańdziuk J., "Monte carlo tree search: a review of recent modifications and applications," *Artificial Intelligence Review*, vol. 56, no. 3, pp. 2497–2562, Mar 2023. [Online]. Available: https://doi.org/10.1007/s10462-022-10228-y

[42] Arenz O., "Monte carlo chess," B.S. thesis, Technische Universität Darmstadt, 2012.

[43] Chaslot G., "Monte-carlo tree search," Ph.D. dissertation, Maastricht University, 2010.

[44] Chaslot G., Saito J.-T., Bouzy B., Uiterwijk J., and Van Den Herik H. J., "Monte-carlo strategies for computer go," in *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium*, 2006, pp. 83–91.

[45] "Tic-tac-toe," https://en.wikipedia.org/wiki/Tic-tac-toe, [Accessed 14-01-2024].

[46] “Tic-tac-toe,” http://gamescrafters.berkeley.edu/games.php?game=tictactoe, [Accessed 14-01-2024].

[47] Çalışkan İ., “Mangala bir zeka oyunu ve kültürümüzdeki yeri,” *Türk Yurdu Dergisi*, vol. 276, pp. 68–73, 2010.

[48] “Mangala | Nasıl | Oynanır — mangala.com.tr,” https://www.mangala.com.tr/mangala-nasil-oynanir, [Accessed 26-12-2023].

[49] “Checkers | Rules, Tactics & Variations — britannica.com,” https://www.britannica.com/topic/checkers, [Accessed 26-12-2023].

[50] “English draughts - Wikipedia — en.wikipedia.org,” https://en.wikipedia.org/wiki/English_draughts, [Accessed 26-12-2023].

[51] Liu A. and Yang Y., “Adversarial search ii,” Apr 2024.

CURRICULUM VITAE

PERSONAL INFORMATION

Name Surname : Emre YILMAZ

Date of Birth :

Phone :

E-mail :

EDUCATION

High School : Kızılcahamam Anatolian High School

Bachelor : Kocaeli University

Master Degree : Ankara Yıldırım Beyazıt University

WORK EXPERIENCE

Aselsan: 2021 - Current

Simsoft Bilgisayar Teknolojileri: 2017-2021

TOPICS OF INTEREST

- Model Driven Development
- Image Processing
- Aviation software development
- Machine Learning