4-5-2021

The Exploration and Analysis of Mancala from an AI Perspective

Trevon J. Hunter
Andrews University, trevon@andrews.edu

J.N. Andrews Honors Program
Andrews University

HONS 497
Honors Thesis

The Exploration and Analysis of Mancala from an AI Perspective

TJ Hunter

4/5/2021

Advisor: Dr. Rodney L. Summerscales

Primary Advisor Signature: [illegible]

Department: Computing

Table of Contents

1. Abstract

Through the study of popular games such as Chess and Go, countless artificial intelligence (AI) research has been conducted in an attempt to create algorithms equipped for adversarial search problems. However, there are still a plethora of avenues that offer insight into further development. Mancala is traditionally a two-player board game that originated in the East and offers a unique opponent-based playing experience. This thesis not only attempts to create a competitive AI algorithm for mancala games by analyzing the performance of several different algorithms on this classic board game, but it also attempts to extract applications that may have relevance to other "game-solving" AI problems.

2. Introduction

This thesis focuses on artificial intelligence, or AI, as it applies to competitive, adversarial games. More specifically, two-player board games. The application of AI to board games is a fairly recent field of study that first gained traction in the 1950s with the development of an artificial intelligence agent for the game of Chess [12]. In its early stages, the AI programs were only advanced enough to compete at a beginner level or exclusively solve games that were in their final few moves before completion. However, over time with continued research and development in this field, the pertinent algorithms have evolved to the point where they can compete and succeed against some of the most advanced players. The culmination of AI game research was exemplified by the defeat of the world's leading Go champion by a Google-developed AI program called AlphaGo [12].

Games such as Checkers, Chess, and Go have been heavily researched in the artificial intelligence community not only due to their popularity but as a result of the underlying complexity behind these games as well. With these various board games, the main goal behind the research is to develop comprehensive algorithms that are able to solve these games without being too computationally expensive. As a result of studying these games, several advancements in the field have been made, although there exists a vast array of other two-player games that have yet to be researched to the same extent. Through further research of other board games that each have their own unique characteristics, it may provide insight into other avenues for potentially improving existing AI algorithms or developing new ones. One such game that has yet to be further researched further is the board game of mancala.

Mancala is a board game that has been around for hundreds of years and is played all around the world. The origins of the word mancala stem from the Arabic word *naqala* which translates to 'moved'. Many individuals commonly associate mancala with one specific board game although the term refers to a family of board games in which there are several different variations. Its origins can be traced back to Ancient Sudan or Ghana with the earliest reliable evidence of the board game being played dating back to around 3600 years ago [7]. Over time, the game has developed with there currently being more than 800 variations of the game played in around 99 countries with about 200 of those variations being designed in more recent times [13]. The variation of mancala that this research will explore is called *Kalah,* and it was developed fairly recently in 1940 by Willie Julius Champion Jr. This version proves to be the most popular in the states and is the version that many westerners commonly associate with the name of Mancala.

2.1 How the game is played

The game (Kalah) is played on a board that has a certain number of small pits, called houses, on each side (usually 6) and a big pit, called the end zone, at each end. A visualization of the board is shown in Figure 1. In each of the houses are a number of seeds (typically 4). The objective of the game is to capture more seeds than your opponent. During a turn, a player grabs all the seeds in a hole on their side and drops them one by one in the succeeding holes following a counter-clockwise pattern until they run out of seeds in their hand. The player deposits the seeds in any hole on the board with the exception of the opponent's end zone. If the last seed dropped lands in the player's end zone, then the player can take an additional turn. If the last seed dropped lands in an empty house on the player's side, and if the opposite house contains seed, all the seeds in the pit where the last seed was placed and all the opponent's seeds on the opposite side go to the player and are placed in their end zone. When a player does not have any more seeds in their pits, the game ends and the opposing player can take all the remaining seeds and place them in their own end zone. The player with the most seeds in their end zone wins.

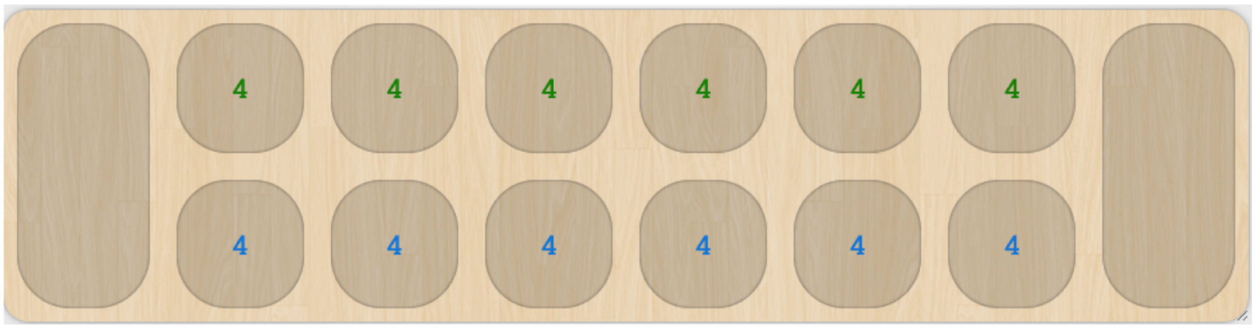

Figure 1: Mancala board

The main reason that mancala was chosen for this research was due to the fact that it is not as widely researched in the artificial intelligence community as some of the aforementioned games such as Chess and Go. Mancala offers a unique playing style that differs from other board games. As a result, the algorithms used on these games would need to be modified slightly, which in turn could provide breakthroughs and new information on how to better improve these respective algorithms. One of the unique aspects of mancala is its multiple turn variation. With traditional board games, the first player usually makes their move, and then the next player makes their move and vice versa. However, with Mancala, as stated in the rules above, if a player is able to drop their last seed in their own store, then they have the opportunity to play again. They are then able to chain multiple moves consecutively, with some players even gaining the ability to go for 4 or more consecutive turns. Another unique feature of the game is its capturing aspect which is dependent on the player having an empty space on their side of the board and calculating their moves so that the last stone dropped lands in that space. Another important motivation for researching the game of Mancala is the fact that its rules are fairly straightforward which makes it fairly easy to translate to a computer version of the game.

The purpose of this research was to develop a competitive artificial intelligent algorithm for this particular game of mancala. The goal wasn't to create an AI that was fun to play with, but rather, create an AI that performs the best at this game, and through the process develop techniques that could possibly be used for solving other similar problems. This research attempts to do so by analyzing the performance of different algorithms on this game. Solving these problems at a more basic level, such as simple two-player board games, gives insight into possible applications for more complex problems, such as multiplayer games, by using the techniques that are formed through the exploration of artificial intelligence at this level.

3. Related Work

As aforementioned, research regarding the intersection of artificial intelligence and gaming is a fairly recent field of study in comparison to other areas in computer science. Over the last few years, researchers have been able to develop more refined algorithms as a result of repetitively applying and adapting these programs to work with a certain set of games. This research project analyzes the performance of six different AI algorithms on the game of mancala. Namely, a random agent, a max agent, minimax with alpha-beta pruning, minimax with an advanced heuristic function, Monte Carlo tree search, and an asynchronous advantage actor-critic agent (A3C). The logistics behind each of these algorithms will be discussed in further detail in a later section, but one of these things that this research aims to do is to extract applications from the analysis of these algorithms that can then possibly be used to develop a new algorithm to refine these existing algorithms.

All of these different AI solving techniques are heavily researched as there is a large body of work backing the six distinct programs mentioned above. Some interesting research projects of note that relate to this thesis are the work by Babaeizadeh et al. [2], regarding the implementation of the A3C agent on a GPU, the work of Wright [20] in regards to the use of a genetic algorithm for parameter optimization -- a methodology that was very crucial for the development of the advanced heuristic function, and the work of Jeerige et al. [9] that further explored the use of the A3C agent for intelligent game playing. In fact, one related work in particular, research performed by Browne et al. [3] on the Monte-Carlo tree search algorithm, is an excellent example of how applying existing AI algorithms to games in different contexts can sometimes yield unexpected insight. The Monte-Carlo tree search methodology was developed as an extension of existing tree search algorithms in order to tackle the complicated game of Go.

In terms of the particular game of mancala, as previously stated, the application of artificial intelligence to the game isn't as heavily researched as some of the other popular board games. However, there are a few prominent works that relate to the game of mancala and its AI applications. The variation of mancala that this thesis analyzes, Kalah, has actually been fully solved in terms of finding the optimal moves for each succession of the game. Irving and Donkers [8] were able to prove that Kalah is a win by 10 for the first player given perfect gameplay from both opponents. In order to track all of the different combinations of moves, the researchers were required to use a full game database and optimized tree-search algorithms. In contrast to the work of Irving and Donkers, this thesis attempts to create an agent that performs optimally on the game of mancala without the use of a full-game database. This research also references the work of Gifford et al. [6] as it explores the approach of AI within the game of mancala strictly from a perspective that employs minimax with a robust emphasis on different heuristic options. Although this research by Gifford doesn't employ more complex algorithms such as Monte-Carlo tree search or an A3C agent, this work served as a good basis for establishing the heuristic approach that was later used in this thesis.

4. Problems with Adversarial Search

Adversarial search is a set of problems within the larger field of artificial intelligence where there is an "enemy" or "opponent" that is constantly changing the state of the problem with every step in a direction that is in contrast to your desired goals [16]. Each agent needs to consider the action of the other agent and the effect of that action on their performance. These types of problems can be seen demonstrated in all areas of life ranging from board games like chess to business situations, trading, and is also the category of search problems under which mancala lies. In order to understand the AI algorithms associated with adversarial search and solving the game of mancala within this research project, it is important to understand how these problems are modeled from the perspective of the computer. These problems are modeled in such a way that the first player (the computer) can change the current state, but is not in control of the next state, i.e. the move that the opponent makes after the first player. The opposing agent, or the opponent, controls the next state and can either change it in a way that is either unpredictable or optimal for them and hostile for the first player. Usually, in these types of problems, you only get to change every alternate state, although it can vary depending on the type of game that you are playing. The reason these games fall under the category of search is due to the way in which a computer operates. In order for a computer to break down these types of problems, the games are usually modeled as a search problem with a heuristic evaluation function. Using this specific modeling, the computer then essentially searches through a broad spectrum of possible outcomes for a solution most advantageous to it based on the aforementioned heuristic evaluation function. The way that these games are usually modeled is through the use of a tree structure where the nodes of the tree are the game states and the edges of the tree are the moves by the players. To provide visualization of this construct, an example of the tic tac toe game state represented in a tree structure is shown in Figure 2 where each of the nodes represent the possible moves for each of the players.

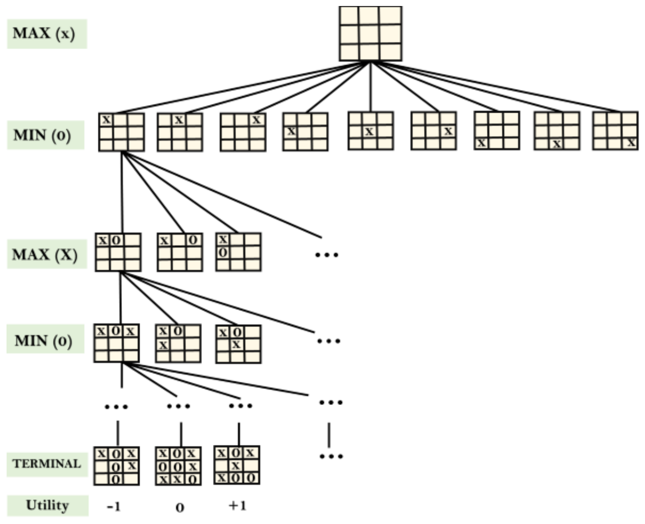

Figure 2: Example of a game tree representation

These games can also be modeled using a structure known as neural nets, where there are several layers of input and an overall outcome corresponding to a win or loss in a game, although modeling these types of problems with a game tree is more common. The first step of the process of solving a game comes down to effectively generating the decision tree or neural network associated with that game that clearly encapsulates all the possible moves and scenarios associated with those moves. A game is considered solved when the programs are able to predict the results of the game from a certain state when all the players make optimal moves. The next step of the process is then to apply different complex algorithms to these structures in order to extract the desired moves that ultimately lead to a win or a maximization of points based on the heuristic function associated with that algorithm.

The main problem with adversarial search stems from the fact that the state space of several games is too large to represent in a tree structure. Attempting to search all the different possibilities of these games would be too computationally intensive. Take for instance the game of chess. On average the player has about 31 to 35 legal moves at their disposal for each turn which would result in about 10^{120} possible games as a conservative estimate [17]. This would be impossible for a single computer to process in an efficient amount of time. Although mancala isn't as computationally taxing as chess, which is another reason that makes it ideal for this research, it would still be inefficient to attempt to solve the entire state space without the use of a full game database. The average branching factor of a game of Kalah is 6 meaning that a decent estimate for game tree complexity would be about 1.74×10^{13} [8]. Although not as complicated as chess, it would still require a lot of computational power to evaluate all of these different game variations. It is for this reason that the algorithms developed must be adaptive in their approach to solving the game. Although it would be convenient, for most use cases, to know the full picture of the game with all of the different moves and combinations of moves, it wouldn't be easily computable and it would be incredibly taxing to store all of the information.

5. Methodology

This research focused on developing a competitive artificial intelligent agent for the game of mancala, more specifically the variation Kalah, through the use of a refined algorithm developed and tested to work with the game space. In order to create this competitive AI agent, 6 different algorithms were developed and then evaluated thoroughly. First, a playable version of the game was developed from scratch incorporating the game rules and logic. Next, the six algorithms were developed to work with this game. This was accomplished through the representation of the gamespace in a format that the computer would be able to understand. The board was represented as an array with each of the corresponding indexes relating to one of the pits on the board. There was also a *getAvaliableMoves* function that returned all the available pits that a player could play given their current position and the state of the board. There were also functions that defined the actions of scooping up the available seeds in a pit and depositing them in each of the available corresponding pits. Using these functions, algorithms were then able to be developed to work with the game and form a strategy for optimal gameplay via their own unique methods. The algorithms developed will be discussed in the following sections and will be listed according to complexity.

5.1 Random Agent

This algorithm serves as the baseline for the evaluation portion of the project. With the worst expected performance, if a developed algorithm isn't performing well against the random agent, then that usually means that there is some error within the algorithm. The random agent is simply an algorithm that looks at the available choices for each turn and chooses a move at random. The pseudocode for the random agent is shown in Algorithm 1.

Algorithm 1: Random Agent

```
1:  function randomAgent
2:      M = getAvailableMoves(board)
3:      return random.choice(M)
```

```
    end function
```

5.2 Max Agent

The max agent calculates the potential return for each of the available moves during the player's turn and chooses the strategy that results in the highest reward. The reward is based on the number of stones that the player is able to collect in their end zone from that particular turn. The algorithm takes into account the state of the board and the available moves at that turn. When the considered move allows for an extra turn, as chain moves are fairly common in the game of mancala, the algorithm takes that into account by running the maxAgent again recursively for the next consecutive move that the player is able to make. Although this recursive call would make the algorithm more complex, consecutive turns one average don't exceed 3 or 4 when playing. Therefore, the computational complexity of the recursive call can be ignored. The pseudocode for the max agent strategy is shown in Algorithm 2. The algorithm is deterministic, meaning that the result of any one action can be predicted with perfect accuracy since there are no random or unknown variables in mancala. This algorithm expectedly would perform better than the random agent and is indicative of an average player's gameplay as the greedy strategy is usually the strategy that most casual players employ.

Algorithm 2: Max Agent

```
function maxAgent(board, player)
    bestValue = --infinity
    chosenMove = [ ]
    M = getAvailableMoves(board)
    pieces = store(board, player)
    for all m in M do
        if evaluate(m, board, player, pieces) ≥ bestValue then
            bestValue = evaluate(m, board, player, pieces)
            chosenMove += m
        end if
    end for
    return chosenMove
end function

function evaluate(move, board, player, pieces)
    if doMove(board, move) is not terminal
        pieces = evaluate(move, board, player, pieces)
    else
        pieces = store(board, player)
    end if
    return pieces
end function
```

5.3 Minimax with Alpha-Beta Pruning

This algorithm is based on the conventional minimax algorithm with the incorporation of alpha-beta pruning in order to minimize the computational complexity by not having to expand as many nodes within the tree. This particular method of solving games was briefly discussed in section 4 and deals with the representation of the game state as a tree structure in order to solve the game. The tree represents the different moves and states of the board, with one player choosing moves to maximize the overall score and the other player choosing moves to minimize the overall score. From the root node, the algorithm takes into consideration all the available moves and plays out each of them individually which in turn leads to a new state of the board. Oftentimes there are scores associated with each respective state of the board to help the computer differentiate which moves are profitable and which ones are not. The associated score with each of the nodes on the tree is based on an evaluation function that takes into account how the game is played. For example, in a simple game such as tic tac toe, moves that result in two of the same player's pieces being next to each other would be rated higher than a move in which the pieces were randomly placed on the board. Ideally, the computer would be able to expand the entire tree and find out the sequence of moves that would result in a win, but as stated in section 4, games like mancala have too many moves and variations for a computer to be able to feasibly search all of the different variations. The reason why an evaluation function is used is due to the fact that the computer can't search the entire tree, therefore it has to estimate which moves would be profitable which is done through the association of the evaluation function. The function that was used for this particular algorithm was simply the number of seeds in the player's bin vs. the number of seeds in the opponent's bin. The maximum depth of tree searched was a depth of 4 to conserve computational efficiency, with the exception of a depth of 8 that was used to compare the minimax with alpha-beta pruning algorithm to the advanced heuristic minimax that will be discussed in the following section.

Alpha-beta pruning is the method of reducing the computational toll of the traditional minimax by limiting the number of nodes that are searched by the algorithm. It does so by updating a value called Alpha which for all moves associated with the main player. As the algorithm progresses through the tree if the evaluation for a particular node is higher than Alpha, then Alpha is updated for that value. Similarly to the main player, the opponent updates a value called Beta and it keeps track of the lowest value returned from the evaluation for each node as the algorithm progresses. The algorithm constantly checks if Beta is less than or equal to Alpha and if that is the case, then all the following branches are skipped as their moves are of no interest to the main player. The pseudocode is outlined in Algorithm 3.

Algorithm 3: Minimax with Alpha-beta pruning

```
1:  function alphaBeta(node, $\alpha$, $\beta$, player, opponent, depth)
2:      if node is terminal or depth = 0 then // leaf node
3:          return evaluate(board, player, opponent)
4:      end if
5:      if whoIsPlaying(board) = player // player playing: maximize
6:          bestValue = --infinity
7:          children[ ] = children(board)
8:          for all child in children do
9:              value = alphaBeta(child, $\alpha$, $\beta$, callingPlayer, depth-1)
10:             bestValue = max(bestValue, val)
11:             $\alpha$ = max($\alpha$, bestValue)
```

```
                if $\beta \le \alpha$ then //pruning
                    break
                end if
            end for
            return bestValue
        else // opponent playing: minimize
            bestValue = *infinity*
            children[ ] = children(board)
            for all child in children do
                value = alphaBeta(child, $\alpha$, $\beta$, callingPlayer, depth-1)
                bestValue = min(bestValue, val)
                $\beta$ = min($\beta$, bestValue)
                if $\beta \le \alpha$ then //pruning
                    break
                end if
            end for
            return bestValue
        end if
    end function

    function evaluate(board, player, opponent)
        return store(board, player) - store(board, opponent)
    end function

    function children(board)
        M = getAvailableMoves(board)
        for all m in M do
            child = doMove(board, m)
            add child to children
        end for
        return children
    end function
```

5.4 Advanced Heuristic Minimax

The advanced heuristic minimax is similar to the alpha-beta minimax in the sense that it uses that generic minimax formula at its core. However, it does differ in one key factor and that is the more refined heuristic function that it incorporates for the evaluation function. This heuristic function and process of refining it is based on the work of Divilly et al. [5] although their work dealt mainly with the mancala variations of Awari, Oware, Vai Lung Than, and Érhérhé. The heuristics that were explored for this particular research were the following

- **H1: Hoard as many seeds as possible in one pit.** This heuristic attempts to keep as many seeds as possible in the leftmost pit on the player's side. The incentive to hoarding pits on the player's side is that at the end of the game after the opponent's side is cleared, all the remaining seeds on the player's side are awarded to them. According to the work of Gifford et al. [6], the leftmost pit is denoted to be the most advantageous pit in which to hoard the seeds

- **H2: Keep as many seeds on the player's side.** This heuristic is a generalized version of the previous heuristic. Rather than attempting to limit the accumulation of seeds to just one pit, it aims to just collect seeds in any of the pits in hopes that they may all be contributed to the main player at the end of the game.
- **H3: Have as many moves as possible from which to choose.** This heuristic takes into account the possible moves that the player might take at each turn and weights the benefits of having more moves to choose from for each turn. It has a look ahead of one.
- **H4: Maximize the number of seeds in a player's own store.** This straightforward heuristic attempts to maximize the number of seeds that the player captures with a look ahead of one that compares the previous amount of seeds in the store to the current amount.
- **H5: Move the seeds from a pit closer to the opponent's side.** This heuristic weighs the benefit of moving the seeds in the far-right pit, from the perspective of the main player, given that it has seeds. If this pit is discovered to be empty then the next rightmost pit is checked and so on. This strategy was originally outlined in the work of Jordan and O'Riordan [10] when discussing certain moves that gave the player a winning advantage.
- **H6: Keep the opponent's score to a minimum**. This heuristic has a lookahead of two moves and tries to limit how much the opponent can score with the next move following the main player's successive move. It takes into account how many seeds are added to the opponent's score with their move. This heuristic has a negative value associated with it as the main player tries to limit this from happening in great quantity.
- **H7: Maximize repeat turns.** This heuristic attempts to maximize the chain moves that are performed by the main player, therefore it prioritizes moves in which the player deposits the last seed into their own store.
- **H8: Points difference.** This is the simple heuristic that was incorporated into the alpha-beta pruning minimax. It simply takes the difference between the main player's store and the opponent's store.
- **H9: How close the player is to winning**. This heuristic captures the research of Gifford et al. (10) which states that if a player reaches 1 and a half the amount of stones of the opposing player, then they are guaranteed to win. The minimum amount of stones that the opponent would have for this condition to be true was set to 5 as the aforementioned research didn't outline this minimum. Once again the heuristic works by simply comparing the number of stones in the main player's store to the number of stones in the opponent store. This has a look ahead of one.
- **H10: How close the opponent is to winning**. This heuristic is simply the inverse of H9. It compares the number of stones in the opponent's store with the amount in the main player's store and checks to see how close the opponent is to having 1 and a half the number of the player's stones. There is a negative value associated with this heuristic to discourage this from happening in-game.

The formula for incorporating the heuristics is simply a linear function that adds up each of the heuristics multiplied by their associated weights. An example of the equation would be computed as:

$$V = H1 \times W1 + H2 \times W2 + H3 \times W3 \ldots\ldots\ldots H9 \times W9 + H10 \times W10$$

The heuristics were first tested in a round-robin, tournament-style setting in order to determine which heuristics were naturally the strongest and which were associated with a higher chance of winning. The heuristics were then associated with temporary weights with the better-performing heuristics being given a higher weight. After these temporary weights were assigned, a genetic algorithm was then used to fine-tune the weights and find a strategy that incorporated all of the various heuristics. The genetic algorithm used a real number representation for the fitness score and ran for 250 generations with a population size of 50. The mutation rate was set to 0.1 and selection was based on their fitness score. A gaussian mutator was used. The fitness score of a candidate was based on how they competed against the rest of the population. The candidate in question with its unique set of weights played 5 games going first and 5 games going second against the entire population including itself. 1 point was received for a win, 0.5 for a draw, and 0 for a loss. The fitness value returned was the percentage of the points received out of all the points that were available to be won. Although the genetic algorithm didn't converge a set of weights was finally settled on which is outlined in the Results section 6. The pseudocode for the evaluation portion of this algorithm is outlined in Algorithm 4.

Algorithm 4: Advanced heuristic minimax

```
1:  function evaluate(board, player, opponent)
2:      H1 = stonesInLeftPit(board, player)
3:      H2 = stonesInPits(board, player)
4:      H3 = numberOfNonEmptyPits(board, player)
5:      H4 = store(board, player)
6:      if doMove(board,player) is rightmost then
7:          H5 = 1
8:      else
9:          H5 = 0
10:     end if
11:     H6 = - store (board, opponent)
12:     if doMove(board, move) is not terminal
13:         H7 = 1
14:     else
15:         H7 = 0
16:     end if
17:     H8 = store(board, player) - store(board, opponent)
18:     if store(board, opponent) ≥ 5
19:         H9 = - (store(board, opponent) * 1.5) - store(board, player)
20:     end if
21:     if store(board, player) ≥ 5
22:         H10 = (store(board, player) * 1.5) - store(board, opponent)
23:     end if
24:
25:     return H1*W1+H2*W2+H3*W3+H4*W4+H5*W5+H6*W6+H7*W7
26:             H8*W8+H9*W9+H10*W10
27: end function
```

5.5 Monte Carlo Tree Search

Monte Carlo Tree (MCT) search is a unique algorithm that was developed as an alternative to the minimax algorithm as a tree searching algorithm for game development. As opposed to the minimax algorithm MCT search does not need a heuristic function. The tree is established in the same fashion as minimax in the sense that each node represents the state of the game and the different branches of the tree represent respective moves. The main difference however is that rather than receiving an evaluation function at each node, MCT search solely takes in the following information from the node: if it's a terminal state, the available moves, and if it is a terminal state information about which player won. The algorithm then uses this info to play out simulations of the game. Starting from the root node it chooses moves until it reaches a terminal state. The process of choosing moves can be more or less random but at its base, there is a strategy to picking the moves that incorporate balancing between moves already played and moves never played. This is also called the process of exploitation and exploration. The result of the game, whether it is a win or a loss is then backpropagated to the root node. After the algorithm plays out the set number of simulations, it then chooses the best move.

The development of this algorithm was based on the work of Moghadam [15] and the research of Chaslot [4]. The computational budget was set as the number of iterations with the maximum number of iterations being 4000. During the selection process, the nodes are picked using an upper confidence bound for trees (UCT) formula with a constant $C_p=\sqrt{2}/2$. The UCT formula only takes into account the score of the player that is playing at that node. The reward that is backpropagation to the root node is a vector containing the score of each of the players for that particular gameplay. The value associated with a victory is 1 and likewise, the value associated with a loss is 0. Once the algorithm is completed, the incoming move that is selected is the most visited node. In terms of the expansion step, a combination of a random and greedy strategy was used known as the epsilon-greedy strategy where at each turn the algorithm has a probability associated with the fact that it plays randomly, ε, otherwise it chooses the moves based on a greedy strategy.

5.6 Asynchronous Advantage Actor-Critic Agent

The implementation of the Asynchronous Advantage Actor-Critic (A3C) agent was made possible through the use of the python libraries Tensorflow and Keras and was based upon the implementation of Julani [11]. The A3C agent is a fairly newly developed method for deep reinforcement learning. Traditionally for reinforcement learning involving game development, Deep Q Networks (DQNs) which heavily revolved around the discovery and refinement of Q values -- values that denoted an associated reward with each action, were used. The A3C agent, however, was recently developed and popularized by Google's DeepMind team. Its success can be attributed to the fact that it differs from the DQN by combining aspects of Q-value learning from DQNs with a policy gradient in order to take advantage of both styles of reinforcement learning. Other work that was referenced during the creation of and training of this agent was the work of Mnih et al. [14] and the work of Alp and Guzel [1].

An A3C agent is comprised of an actor and a critic. The actor is responsible for choosing the actions that the main player takes and the role of the actor is to control the behavior of the player by learning the best policy. The actor takes a specific state as the input and outputs the best action. It does so by using the feedback that it receives from the critic. The critic learns a value function that represents the expected value from each state based on the specific action. It uses this value function to determine how advantageous it is to be in a particular state. Over time the actor and the critic both become refined to the point where they have a pretty good working policy for the gameplay.

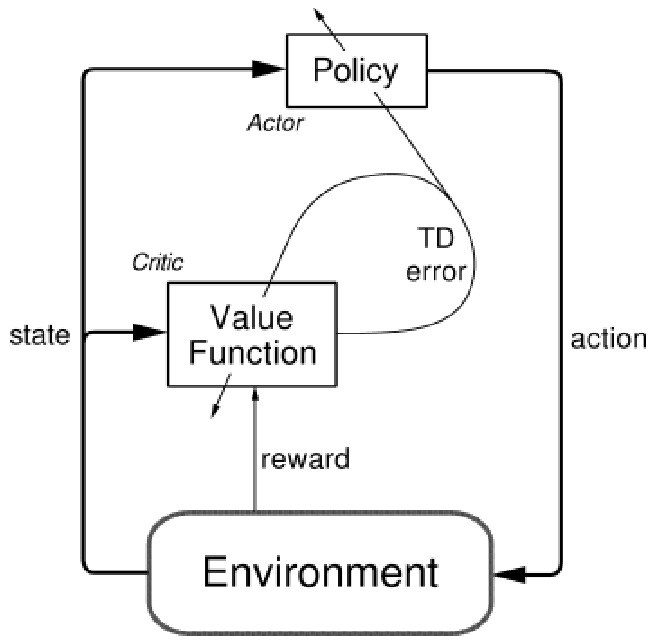

Figure 3: The actor-critic architecture. From Sutton & Barto [18]

The A3C agent differs from the DQN in the sense that it doesn't need to learn the Q values for each state which in turn saves its calculation time allowing for a more robust algorithm. Instead of the Q value, the advantage is used which is denoted as the difference between the actual return at a specific state and the predicted value which is derived from the value function. The advantage is how the critic can tell if its predicted value is good or bad. The advantage is an estimation function that is commonly written as:

$$A(s) = r+\gamma V(s') - V(s)$$

where r is the current reward and γ is the discount factor. The critic knows the value of the state but doesn't know how much better the value returned from the value function is. This is where the advantage comes into play. The higher the advantage, the more agents will look at doing the action.

The actor and the critic loop through each step of the game, or in the cases of mancala the different board states, and update the weights and policy accordingly. The workflow for this agent typically looks like this. The worker (actor) takes in the parameters, the value function, from the critic. Using these values it receives from the critic, it then updates its probability distribution and interacts with the environment. The worker then calculates the value and policy loss at the end of the episode. The worker then gets gradients from the loss. Lastly, the worker updates the global networks with the gradients. With the A3C agent, this all takes place asynchronously and oftentimes there are multiple workers that are approaching the environment while being initialized differently meaning that they are dealing with different states of the game. Although having several workers being trained asynchronously is a little bit more computationally expensive, it does speed up the overall process. These workers are all solving the environment in different ways and updating the global network.

After the global network is updated, the different workers are then also updated in turn from the global network with the updated value functions which in turn update the policy. Each of these workers, however, are doing independent exploration and training before they update therefore the updates are not happening simultaneously. This is one of the drawbacks to the asynchronous nature of the A3C agent. It is quite possible that the different agents are playing with older versions of the parameters.

From the perspective of neural networks, the agent gives two outputs: the value and the policy. The value output is a function that represents the sum of rewards when starting in state s and following the policy. The policy output is a vector that represents the probability distribution over all of the actions, or rather the probability to select each action. The actions that are performed by the neural network are chosen non-deterministically based on the probability that they will be selected. Based on these two separate forms of output we arrive at two loss functions for the neural networks which were automatically implemented into the code architecture. The goal is to minimize these loss functions. (R represents the discounted future rewards). The value loss is a simple sum squared error which is represented as

$$L = \Sigma(R - V(s))^2$$

with R signifying the discounted future rewards. The policy loss is a logarithmic function that's represented by

$$L = -\log(\pi(a \mid s)) * A(s) - \beta * H(\pi).$$

$H(\pi)$ is the entropy and is simplified to the function

$$H(\pi) = -\Sigma(P(x) \log(P(x))).$$

The entropy represents how spread out the probabilities are and incorporating it into the equation limits the chance of the policy converging to a local optimum. The two loss functions are then combined to get a single loss function for the model overall. The equation is represented as

$$L = 0.5 * \Sigma(R — V(s))^2 - \log(\pi(a \mid s)) * A(s) - \beta * H(\pi).$$

As is evident from the equation, the value loss is set to 50% in order to put more emphasis on policy learning as opposed to value learning.

As aforementioned in the workflow of the A3C agent, training of the two networks is performed separately and gradient ascent, as opposed to gradient descent is then used to find the global maximum and update both their weights. Some key components of the code architecture were (17):

- **AC_Network**: The class containing all the Tensorflow ops to create the networks themselves
- **Worker**: The class containing a copy of AC_Network, an environment class, as well as all the logic for interacting with the environment and updating the global network
- **High-level code** for establishing the worker instances and running them in parallel

5.7 Closing Methods

After these respective algorithms were developed and refined, they were then evaluated and compared through a number of tests and simulated gameplay. All development for these aforementioned algorithms was done in python.

6. Results

Every combination of algorithms was tested for 1000 simulated games where they played against each of the developed algorithms including themselves. The results demonstrate the strong first move advantage that exists in the game of mancala. The performances of the respective algorithms were also roughly in line with the algorithm's complexity. The results of the algorithm matchups are demonstrated in Table 1.

	Random	**Max Agent**	**Minimax (Alpha Beta)**	**Heuristic Minimax**	**Monte Carlo Tree Search**	**A3C Agent**
Random	51.3%	4.1%	2.7%	0.5%	0%	4%
Max Agent	96%	55.3%	15.2%	12.5%	4.5%	26.4%
Minimax (Alpha Beta)	98.2%	85.8%	61.5%	35.2%	N/A	48%
Heuristic Minimax	99.5%	90.2%	68.5%	61%	N/A	53.5%
Monte Carlo Tree Search	100%	87.4%	59.3%	44.7%	61.3%	N/A
A3C Agent	97.6%	89.3%	71.4%	67.1%	N/A	79%

Table 1: Table comparing the win percentages of each player

The table is organized in a structure where the agent playing as the first player is in the left column and player 2 is represented in the top row. Going across each of the rows shows how many games the specific algorithm won playing as the first player. Going down each column shows how many games the algorithm lost going as the second player. The win percentages were based on the percentages of games won out of all the simulated games that were played (1000). The remaining percentage that is not accounted for in the algorithm's win percentage doesn't necessarily signify a loss but could also account for draws, which is fairly common in the game of mancala.

Based on the figure we can see that the A3C agent appears to have the overall higher win percentages, albeit by a small margin in comparison to the Monte Carlo tree (MCT) search and the heuristic minimax, while the random algorithm expectedly performs the worst. The significant trend exemplified in the table is the fact that the win percentages go up based on the complexity of the algorithm. The heuristic minimax, MCT search, and A3C agent all perform at around the same level as each of them, differentiating levels of success against various algorithms without one completely outperforming the other. For example, although the A3C agent has higher win percentages against the max agent and the alpha-beta pruning minimax, it does not perform as well against the random agent as the heuristic minimax and MCT search.

The minimax algorithms weren't compared to the MCT search in a manner that could show a single result for the win percentage. Rather, they were compared iteratively showing the relationship between the number of iterations for MCT search and the depth of the minimax algorithms in order to establish thresholds for both algorithms. As aforementioned, the figures that were eventually settled on were 4000 for the number of iterations and 4 for the depth. This comparison between the minimax algorithms and the MCT search is shown in Figure 4 and Figure 5. In addition, the A3C agent wasn't compared to the MCT search due to the computational load of both of these algorithms.

Lastly, when looking at the table, we can see that the winning percentages for an algorithm playing against itself go up as the complexity of the algorithm increases. The winning percentage should in theory be around 50% as is exemplified by the random agent. However, mancala holds a strong first move advantage meaning that given perfect gameplay from both parties, the player that goes first is guaranteed to win. Therefore, it is shown that as the algorithms become more complex, they take advantage and capitalize on this first move advantage in increasing effect. It is for this reason that the A3C agent has a win percentage of 79% against itself because as the first player it won 79% of the matches. Theoretically, the perfect agent should win 100% of the matches.

As mentioned in section 5.4, the weights associated with the heuristic function were trained using a genetic algorithm. To first establish which weights held the strongest influence over the winning percentages of the game, they were first compared in a round-robin style tournament. Using just one heuristic as the evaluation function, 10 games were played against all the other heuristics. The results of the tournament are shown in Table 2 with the winning percentage being the percentage of games won out of all the matchups that were played.

Heuristic	Win Percentage
H1	56%
H2	44%
H3	0%
H4	100%
H5	33%
H6	78%
H7	89%
H8	67%
H9	22%
H10	11%

Table 2: Table showing the results of the round-robin matchups for the heuristic values

Based on the results of the tournament it was fair to conclude that the strongest heuristics were H4, H6, and H7 with the weakest one being H3. To initialize the genetic algorithm, these heuristics were then given a higher weight than all of the other respective heuristics. The weaker heuristic was also initialized to a lower value. The genetic algorithm ran for 20 different rounds and the results can be seen in Table 3.

Run	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	Fitness
1	0.322	0.124	0.442	1	0.665	0.657	0.776	0.585	0.127	0.263	57.3
2	0.136	0.195	0.671	1	0.561	0.548	0.848	0.676	0.278	0.371	58.4
3	0.356	0.303	0.366	1	0.413	0.669	0.943	0.639	0.153	0.159	59.5

4	0.046	0.286	0.438	1	0.659	0.656	0.741	1	0.116	0.112	59.7
5	0.301	0.163	0.526	1	0.428	0.887	0.984	0.829	0.392	0.399	63.9
6	0.274	0	0.418	1	0.462	1	0.855	0.452	0.391	0.366	60.3
7	0.354	0.197	0.325	1	0.491	0.621	1	0.754	0.272	0.257	61.4
8	0.245	0.285	0.398	1	0.673	0.584	0.969	0.167	0.319	0.378	61.2
9	0.379	0.376	0.714	1	0.632	0.797	0.991	0	0.219	0.159	60.7
10	0.225	0.283	0.576	1	0.644	0.778	0.936	0.987	0.229	0.214	64.9
11	0.148	0.212	0.777	1	0.637	0.629	0.814	0.654	0.128	0.384	62.1
12	0.142	0.081	0.474	1	0.653	0.512	0.715	0.594	0.246	0.247	59.9
13	0.121	0.324	0.434	1	0.466	0.514	0.824	0.632	0.314	0.125	58.3
14	0.021	0.241	0.339	1	0.599	0.642	0.825	0.586	0.261	0.382	60.5
15	0.136	0.078	0.317	1	0.437	0.579	0.768	0.993	0.194	0.375	60.9
16	0.123	0.124	0.314	1	0.413	0.532	1	0.914	0.122	0.339	64.3
17	0.212	0.388	0.393	1	0.664	0.527	0.957	0.467	0.365	0.326	65.2
18	0.336	0.132	0.682	1	0.696	0.667	0.748	0.229	0.111	0.229	68.6
19	0.225	0.122	0.654	1	0.484	0.694	0.918	0.667	0.194	0.297	68.7
20	0.234	0.266	0.465	1	0.415	0.527	0.962	0.332	0.349	0.329	66.2

Table 3: Table showing the weights of the heuristics after 20 iterations of the genetic algorithm

Although the genetic algorithm didn't converge to a specific set of weights, there are some observable patterns with the heuristic weights that are presented in the table. It is clear to see that H4 is one of the most important heuristics as its weight is consistently the highest value or tied for the highest weight. For each iteration, H4 receives a weight of 1 which is the highest value for a weight that a heuristic can receive. Based on the table it is also evident that H7 is the second-highest weighted heuristic. These results for the most prominent heuristic are predictable based on the outcome of the round-robin tournament. H1, H2, H9, H10 are rated rather lowly as heuristics as they can be seen to never have a weight over 0.4. There does exist some correlation between these sets of algorithms as H2 is simply the abstracted version of H1 and H9 and H10 are the same heuristic but for different opponents. H3 on its own was a bad heuristic. It did terribly during the round-robin tournament as is evidenced in the prior section. However, surprisingly in 7 of the rounds, it had a weight over 0.5. Lastly, the table also shows that the weight associated with H8 fluctuates by the greatest degree. Sometimes it is one of the highest and then it will suddenly become one of the smallest from run to run. This is a heuristic that either needs to be modified or omitted in future research as it seems its contribution is inconclusive. With H4 and H7 being the two highest-rated heuristics, evidently, the overall heuristic function leans towards a more offensive strategy

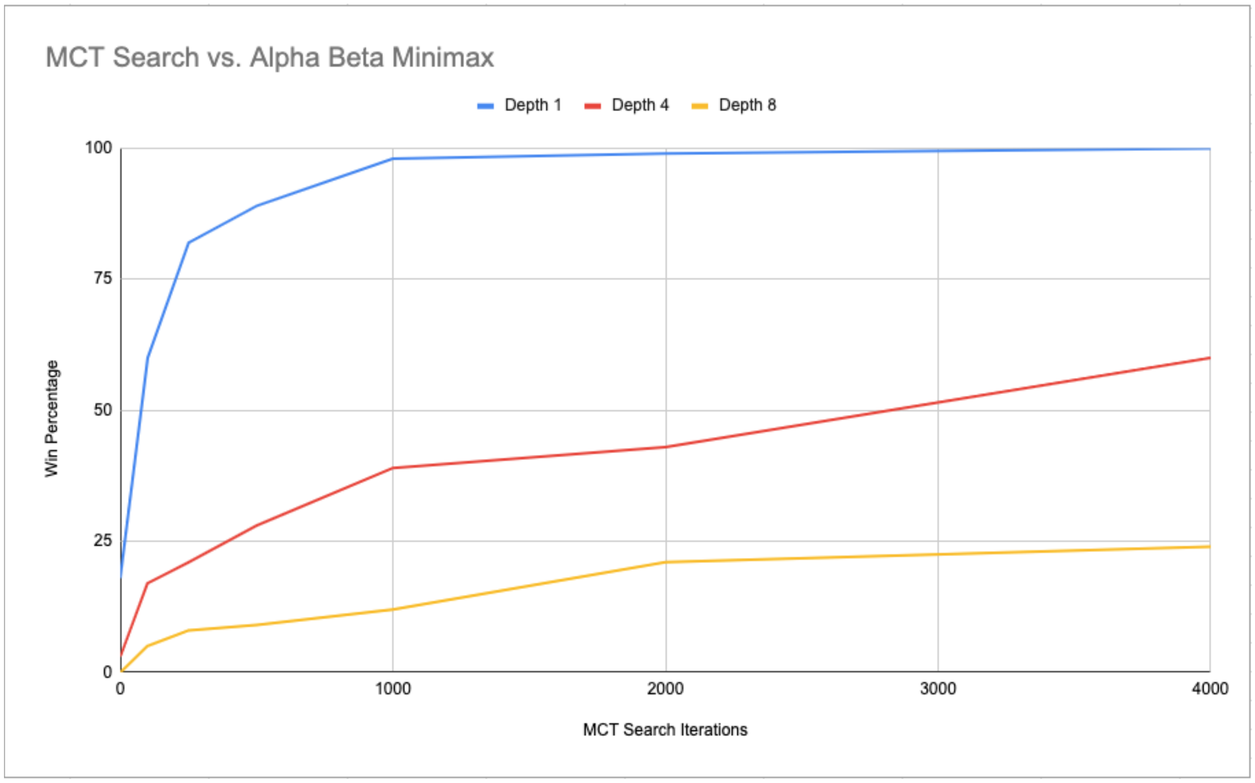

Figure 4: Comparison of MCT search with the Alpha Beta minimax at different depths

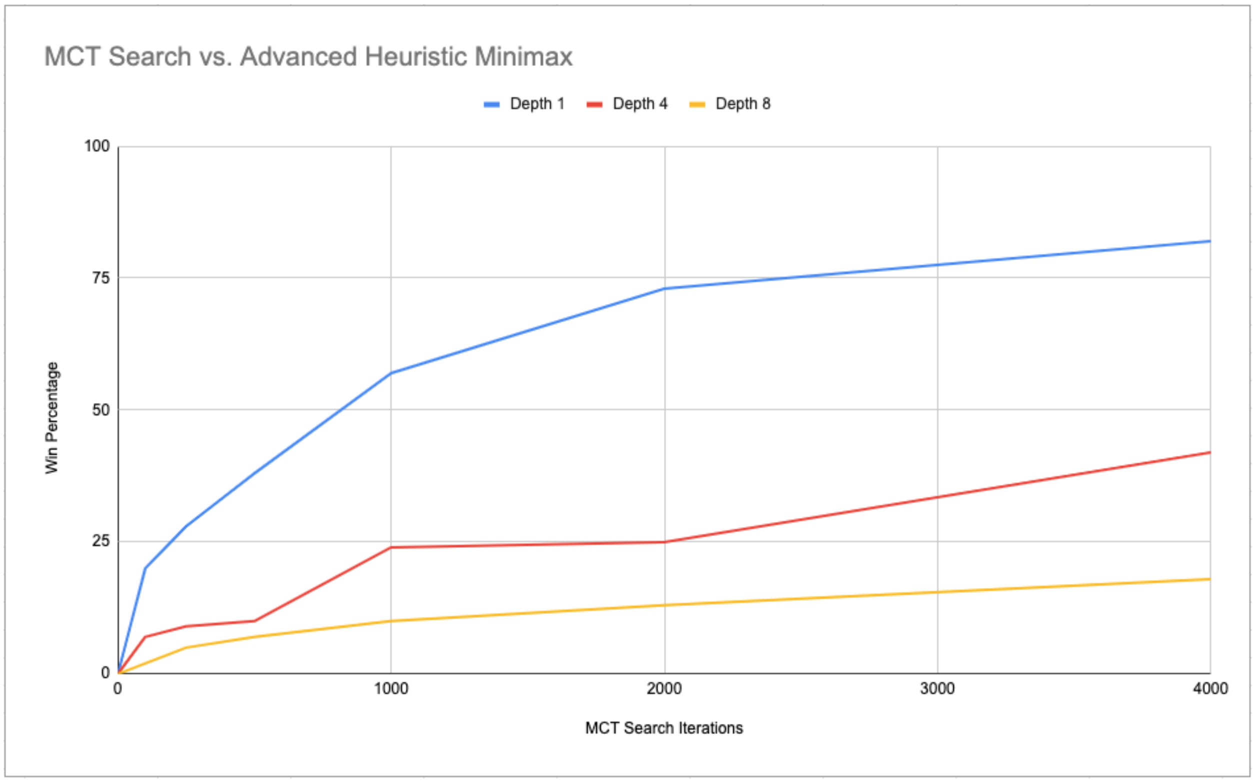

Figure 5: Comparison of MCT search with the Advanced Heuristic minimax at different depths

When analyzing the comparison of the MCT search and the two different minimax functions (represented in Figure 4 and Figure 5), the number of iterations that are performed for MCT search and the depth of the respective minimax algorithms are taken into account. In comparison to the minimax algorithm, MCT search is a fairly computationally expensive agent. By analyzing the performance of MCT search at different numbers of iterations, patterns can then be extracted as to which level of iterations are sufficient for solving the problem as the graph starts to flatten over time. It is for this reason that 4000 was settled on for the number of iterations as the graph began to flatten around this number.

Based on Figure 4 it is evident that the MCT search performs very well against the alpha-beta minimax when it is at depth 1. However, as the alpha-beta minimax algorithm increases in depth, we see that the performance of the MCT search begins to drop. Reaching about 60% win percentage against the algorithm at a depth of 4. It's interesting to see the MCT search doing so poorly against the minimax algorithms especially considering that the MCT search is the only algorithm that was able to get a win percentage of 100% against the random agent. Something that both minimax algorithms were not able to do.

Analyzing the graph in Figure 5, the MCT search expectedly performs worse against the advanced heuristic minimax with the MCT search not even being able to achieve a win percentage of 100% against the heuristic minimax at a depth of 1. It is also shown that at a depth of 4 for the advanced heuristic minimax, the graph of the MCT search win percentages begins to flatten out at around 42%. For both minimaxes at a depth of 8, the MCT search has trouble reaching a win percentage of over 30%. It should also be noted that when dealing with the minimax algorithms at a depth of 8, their computational costs outweigh that of the MCT search.

The last comparison that was conducted was between the alpha-beta pruning minimax at depth 8 and the advanced heuristic minimax at the normal depth for the research, depth 4, in order to see by what margin the advanced heuristic minimax was better than the alpha-beta pruning one. These two algorithms were run for 500 games, and surprisingly the advanced heuristic minimax still outperformed the alpha-beta pruning minimax at a depth of 8 with the advanced heuristic minimax having a win percentage of 54.5%. Although not a large margin, this is still impressive considering that the advanced heuristic minimax is operating at a lower depth.

7. Discussion

While the Actor-Critic (A3C) Agent does appear to have better win percentages, albeit, by a slight margin in comparison to the Monte Carlo Tree (MCT) search and Advanced Heuristic minimax (AHM) algorithm, there are advantages and disadvantages associated with each of the respective algorithms. Although the three aforementioned algorithms all perform comparably well, in terms of overall strength, the Heuristic minimax actually proves to be the strongest algorithm developed for mancala due to its computational efficiency in comparison to the other two algorithms. Once its heuristic was fine-tuned, the AHM algorithm was able to perform considerably well without requiring too much depth. While the A3C Agent is also excellent, it does require loading a large model, expensive training, and fine-tuning hyperparameters. The MCT search on the other hand also plays acceptably well considering it does not have any knowledge of the game nor does it employ any state evaluation functions. However, MCT search requires high numbers of iterations to obtain results, resulting in high computational times. In addition, MCT search performs poorly against the AHM. The research goal of creating a competitive artificial intelligence algorithm for mancala was accomplished. The heuristic minimax algorithm is currently the best algorithm developed for approaching this particular game of mancala due to its robust, and unique heuristic function that enables it to perform competitively well at a lower computational cost than the other algorithms.

The success of the AHM can be attributed to the fact that it took advantage of a greedy strategy, a component that was reflected in its heuristics. H4 and H7 were two of the highest performing heuristics which ended up receiving some of the higher weights as the genetic algorithm evolved. H4 and H7 were two of the main defining characteristics of the max agent, and as is evident from its relative performance mancala is a game that tends to reward a greedy strategy. The only problem is that the max agent's lookahead is only limited to one with the rare exception being in the incorporation of chain moves. AHM in a sense provides the best of both worlds by not only being fine-tuned to incorporate the greedy strategy but also providing a higher look ahead and incorporating other strategies as well.

The random agent and alpha-beta minimax (ABM) both served as excellent baselines throughout the experiment. The random agent was a baseline for all the algorithms as it was expected to perform the worse, therefore it was beneficial to judge the performance of an algorithm based on how it did against the random agent. The ABM algorithm more served as a baseline for the AHM. Although the ABM was competitive on its own and would probably be the implementation used to play as the computer for a commercial version of this game, its main purpose was to comparatively measure the performance of the AHM due to the fact that all of its functionality such as the alpha-beta pruning aspect and evaluation function was incorporated and extended upon in the AHM. The main advantage of the ABM is its ability to be competitive at the lowest computational cost in comparison to all the other complex algorithms.

As stated earlier, the max agent was meant to be representative of average player gameplay. Although the max agent performed poorly against the more complex algorithms, it provided insight into the fact that mancala is a game that is more supportive of an offensive strategy i.e. attempting to maximize your pieces rather than trying to limit that of your opponents. This concept then evolved into the AHM. The main advantage of MCT search is the fact that it requires no prior knowledge or a heuristic function to work. The algorithm also performs extremely well against agents without a concrete strategy such as the random agent but breaks down when going up against more complex agents with refined strategies. The main downside is the computational toll that is required in order to get higher performance. The higher the number of iterations, the better the performance, although the computational expense begins to outweigh the increase in performance at around 4000 iterations.

Lastly, out of all the algorithms, the A3C agent showed the most potential and demonstrated indications that it could possibly outperform the AHM given a few modifications. The main drawback of the A3C is the fact that it is meant to work in a continuous action space. Possibly finding a method to refine it to work with a deterministic action space such as a board game could possibly improve its performance on mancala. As it stands right now, due to the fact that the A3C agent is meant to work with more complicated games, for approaching this process of solving mancala, the A3C agent seemed a bit overkill and was taxing both in the training and development process. As previously mentioned, based on this research the AHM is the best competitive agent for the game of mancala. A possible improvement for the future could be further refinement of its heuristic weights using the genetic algorithm. Although the genetic algorithm is computationally expensive, the advantage for the AHM lies in the fact that the genetic algorithm does not need to be run each time that the AHM is used.

8. Future Work

There are several further avenues for development of this research. The main one being the development of a graphical interface to accompany this project. In this manner, players could select which algorithm they would like to play (organized by difficulty) or they could watch a live animated version of two of the algorithms playing against each other. The use of the command line was sufficient for the development of this research, however, a visual aspect would provide a better connection with those not familiar with how the game works or technical procedures.

Another avenue for future research is the analysis of different variations of mancala. As aforementioned, there are over 800 variations of the game all with different rules and playing styles. To name a few of the popular ones in addition to Kalah there is also Awari, Oware, Vai Lung Thlan, Ohvalhu. Some of these variations also have different board configurations. Applying these algorithms to some of the different variations would provide interesting insight especially since the advanced heuristic minimax was originally created to work with Kalah.

Lastly, as was mentioned earlier, one of the main contributing factors to the success of the AHM was the refinement of the weights which was made possible through the genetic algorithm. Taking into consideration how well the genetic algorithm performed in regards to the refinement of the heuristic and how the A3C agent works it is possible that a combination of the two would yield beneficial results to research on the game. This would look similar to the research applied by Wang [19] in regards to a hybrid variation of a genetic algorithm and a neural network. That being said, both of these agents are very computationally expensive, therefore a method for cutting down the computational cost of the A3C agent could possibly be modifying it to work primarily with a continuous action space to more of a deterministic one, whether this means the incorporation of predefined heuristics or some other method.

9. Bibliography

[1] Alp, E. C., & Guzel, M. S. (2019). Playing Flappy Bird via Asynchronous Advantage Actor-Critic Algorithm. *arXiv preprint arXiv:1907.03098*.

[2] Babaeizadeh, M., Frosio, I., Tyree, S., Clemons, J., & Kautz, J. (2016). Reinforcement learning through asynchronous advantage actor-critic on a gpu. *arXiv preprint arXiv:1611.06256*.

[3] Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., ... & Colton, S. (2012). A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, *4*(1), 1-43.

[4] Chaslot, G. M. J. B. C. (2010). *Monte-carlo tree search*. Maastricht University.

[5] Divilly, C., O'Riordan, C., & Hill, S. (2013, August). Exploration and analysis of the evolution of strategies for mancala variants. In *2013 IEEE Conference on Computational Intelligence in Games (CIG)* (pp. 1-7). IEEE.

[6] Gifford, C., Bley, J., Ajayi, D., & Thompson, Z. (2008). *Searching and game playing: An artificial intelligence approach to Mancala*. Technical Report ITTC-FY2009-TR-03050-03, Information Telecommunication and Technology Center, University of Kansas, Lawrence, KS.

[7] Igwe, E. (2019, May 5). *The history of mancala*. https://www.loyalnana.com/stories-1/2019/5/3/the-history-of-mancala.

[8] Irving, G., Donkers, J., & Uiterwijk, J. (2000). Solving kalah. *Icga Journal*, *23*(3), 139-147.

[9] Jeerige, A., Bein, D., & Verma, A. (2019, January). Comparison of deep reinforcement learning approaches for intelligent game playing. In *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)* (pp. 0366-0371). IEEE.

[10] Jordan, D., & O'Riordan, C. (2011). Evolution and analysis of strategies for mancala games. In *Proceedings of the GAME-ON* (pp. 44-50).

[11] Juliani, A. (2018, June 21). *Simple Reinforcement Learning with Tensorflow Part 8: Asynchronous Actor-Critic Agents (A3C)*. Medium. https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-8-asynchronous-actor-critic-agents-a3c-c88f72a5e9f2#.dgiztjv7l

[12] Kurenkov, A. (2016). A Brief History of Game AI up to AlphaGo. *Andrey Kurenkov*, *18*.

[13] *Mancala*. Mancala World. https://mancala.fandom.com/wiki/Mancala

[14] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., ... & Kavukcuoglu, K. (2016, June). Asynchronous methods for deep reinforcement learning. In *International conference on machine learning* (pp. 1928-1937). PMLR.

[15] Moghadam, M. M. (2021, January 22). *Monte Carlo Tree Search: Implementing Reinforcement Learning in Real-Time Game Player*. Medium. https://towardsdatascience.com/monte-carlo-tree-search-implementing-reinforcement-learning-in-real-time-game-player-25b6f6ac3b43

[16] Russell, S., & Norvig, P. (2002). Artificial intelligence: a modern approach.

[17] Shannon, C. E. (1993). Programming a computer for playing chess. In *first presented at the National IRE Convention, March 9, 1949, and also in Claude Elwood Shannon Collected Papers* (pp. 637-656). IEEE Press.

[18] Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

[19] Wang, L. (2005). A hybrid genetic algorithm–neural network strategy for simulation optimization. A*pplied Mathematics and Computation*, *170(2)*, 1329-1343.

[20] Wright, A. H. (1991). Genetic algorithms for real parameter optimization. In *Foundations of genetic algorithms* (Vol. 1, pp. 205-218). Elsevier.