ANALYSIS OF ARTIFICIAL INTELLIGENCE TECHNIQUES FOR KONANE

by

Kaitlin Hendrick

Submitted in partial fulfillment of the

requirements for Departmental Honors in

the Department of Computer Science

Texas Christian University

Fort Worth, Texas

May 7, 2018

ANALYSIS OF ARTIFICIAL INTELLIGENCE TECHNIQUES FOR KONANE

Project Approved:

Supervising Professor: Michael Scherger, Ph.D.

Department of Computer Science

Antonio Sanchez, D.Sc.

Department of Computer Science

Wendy Williams, Ph.D.

John V. Roach Honors College

ABSTRACT

This research analyzes artificial intelligence techniques for Konane. The game Konane, also known as Hawaiian checkers, is a two-player, zero-sum strategy board game ideally suited for this type of research. In order to have a successful strategy, a player must consider many future possibilities. We compare computing agents that use informed and uninformed searching algorithms but focus our investigation on the effectiveness of the minimax algorithm. By altering variables such as the cutoff depth for searching the game tree and incorporating alpha-beta pruning, we begin to see varying levels of success and efficiency from the competing computing agents. The outcome of this research is an analysis of the effectiveness of each computing agent showing the positive correlation between the depth of the search tree and the percentage of games won and the exponential relationship that exists between the number of nodes explored and the depth of the search tree.

Table of Contents

1 Introduction

Konane, also known as Hawaiian checkers, is a two-player board game where players take turns jumping each other's game pieces. The game ends when a player does not have a move in which they can capture an opponent's piece, and this player would lose the game. At any point in the game, a player can have many choices for a move and the number of game states to examine is exponential in the number of moves. Therefore, it is difficult for a human player to comprehend the future effects of a single move as the game progresses and devise an effective strategy. Konane also satisfies the many contingencies that make it suitable for this type of study in the area of artificial intelligence. We explore designing computer agents to play the game Konane and their effectiveness. We concentrate on implementing two basic artificial intelligence algorithms: the minimax algorithm and a variant called minimax with alpha-beta pruning. The minimax algorithm recursively searches a game tree depth-first, returning a heuristic value of terminal nodes and assigning heuristic values to parent nodes. This value indicates how good a position is for a player in terms of how likely the game state leads to a win for the player. The additional feature of alpha-beta pruning is used to make the minimax algorithm more computationally efficient by reducing the number of game states that need to be explored in order to make a decision. We examine the effectiveness of these computer agents using artificial intelligence techniques to play the game Konane.

1.1 The Game Konane

Konane is a board game dating back to ancient Hawaii. The game is traditionally played on a rectangular, sometimes square, board made out of stone with indentions where the game pieces should go. The size of the game board can range from a 6x6 to a 14x14 including every size

variation in between and even beyond. The game is initially set-up with black and white game pieces, traditionally shells and lava rocks, arranged in an alternating checkerboard pattern over every space of the game board [1]. To begin the game, the first player, typically whoever is represented by the black game pieces, removes one of his pieces from a corner of the board or from the group of four pieces in the very middle of the board. Then, the other player removes one of his white pieces adjacent to the now empty space. For the game board shown in Figure 1, the possible initial moves for black include removing the game piece at (0,0), (3,3), (4,4), or (7,7), using (row, column) notation. If black removes his piece at (0,0), then white can remove either (1,0) or (0,1), whereas if black removes his piece at (4,4), then white can remove (4,3), (3,4), (5,4), or (4,5). For the remainder of the game, players take turns making moves. Every move must jump an opponent's piece in an orthogonal direction and land in an empty space. After a piece is jumped, it is removed from the game board, similar to Checkers. Jumping multiple pieces is allowed so long as the direction of the jump does not change and there is an empty space between each opponent's piece jumped. The game ends when a player does not have a move in which he can capture an opponent's piece, and that player loses the game.

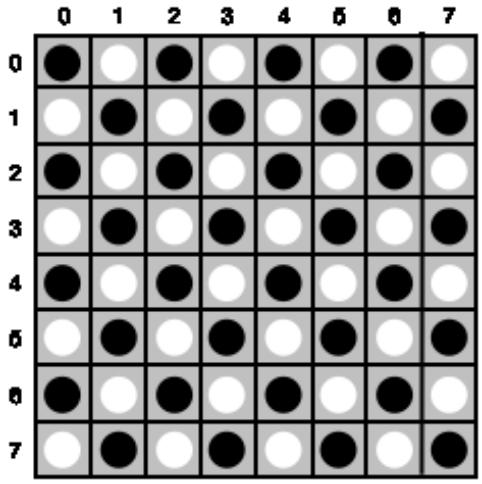

Figure1: Game board set-up at the beginning of Konane.

The number of legal moves a player can choose from is dependent on the stage of the game. In the beginning stage of the game, a player has very few choices of moves because there are not many empty spaces that allow a legal jump move. As more pieces are captured, the number of moves a player can choose from in a single turn increases. Eventually the number of moves quits increasing and remains relatively constant in this middle stage of the game. In the last stage of the game, the number of moves a player has to choose from again decreases as there are fewer game pieces left on the board. Therefore, in this game, a player must consider the potential moves they can make to jump an opponent's piece and be aware of protecting their pieces from being jumped by their opponent. The rules of Konane are very easy to learn and implement but devising a strategy to master the game is quite difficult.

1.2 Artificial Intelligence

There is not one widely accepted definition of what constitutes artificial intelligence (AI). The textbook *Artificial Intelligence: A Modern Approach* shows definitions for artificial intelligence from a variety of sources divided along the dimensions of thought processes and behavior. The definitions are further divided based on measuring against human performance or "an ideal concept of intelligence, which we will call rationality" [2]. All of the definitions share the concept of designing a way for a computer to complete a task involving decision-making. We aim to devise a process for analyzing information and produce an action for the intellectual task of choosing a move in the game Konane. Therefore, we concentrate on the definition from *Computational Intelligence: A Logical Approach* regarding designing intelligent agents [3]. The word agent comes from the Latin word meaning to do [2]. In our case, a player is an agent that understands the game as its environment, and we want to use AI techniques to create an

intelligent agent that chooses the action that maximizes its chance of achieving its goal, winning the game Konane. We do this by implementing the minimax algorithm.

1.3 Minimax Algorithm

The minimax algorithm presents a strategy for choosing a move that will most likely lead to a terminal state that is a win in the game Konane. This algorithm uses an informed, adversarial search to choose the best move given a state of the game and which player's turn it is. This is different from a uniformed search, which has no additional knowledge about the game state, because in this informed search, a static evaluation function assigns heuristic values to states in the game, giving us more knowledge about the future possibilities of the game [3]. Since games are adversarial search tree problems, we must take into account the opponent's response to a player's moves. We can determine the minimax value for a node assuming that both players are playing optimally.

Figure 2 shows a game tree for an arbitrary game. Each node represents a state in the game tree and each line is a move to get to the next state. Terminal states, when the game is over, are marked in red. The minimax algorithm behaves with the assumption that if a move is good for one player, it is bad for their opponent. To this end, the algorithm uses two players, MAX and MIN to simulate the decision of choosing the best move. A triangle indicates a MAX node, which means it is MAX's turn to move, while a circle indicates that it is the MIN player's turn. The rows are also labeled to show this. The numbers inside the shapes represent the utility value of a node after the algorithm calculates these from the static evaluation function. MAX will always choose a move that has a higher utility value while MIN prefers a lower value, as their names describe.

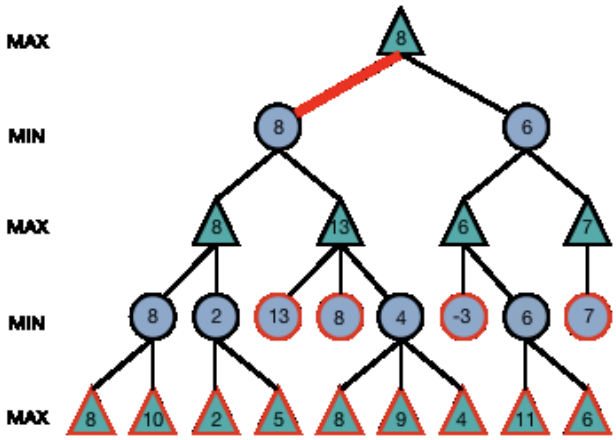

Figure 2: A game tree for an arbitrary game with nodes showing utility values.

The minimax algorithm begins at the current state of the game recursively searching the game tree for the optimal move depth-first, which means exploring a branch as far as it goes and then backtracking until a path that has not been explored is found. Once the algorithm reaches a leaf node, which indicates a terminal state, or reaches the specified search depth in the game tree the utility value of the node is returned. This value represents a heuristic value for the node based on how good a board state is for a player. These values are backed up through the tree and compared to other potential moves' utility values. Depending on which player's turn it is, a node's utility value becomes either the minimum or maximum of all its children node's utility values. The evaluation of the minimax value of a game state is shown in Equation 1 by the MINIMAXDECISION() function, which is passed the game board and which player's turn it is. Move is valid move in the list of potential moves called moves, and NEXTBOARD() creates a copy of the current board and reflects the board as if the move it is passed was executed.

$$\text{MINIMAXDECISION}(board, player) =$$

$$\begin{cases} \text{UTILITY}(board) & \text{if board terminal state} \\ \text{MAXVALUE}_{move \in moves}\ \text{MINVALUE}(\text{NEXTBOARD}(move)) & \text{if board is a MAX state} \\ \text{MINVALUE}_{move \in moves}\ \text{MAXVALUE}(\text{NEXTBOARD}(move)) & \text{if board is a MIN state} \end{cases}$$

Equation 1: Minimax-value evaluation of a node [2].

The game tree in Figure 2 shows the utility values for every node as a result of the minimax algorithm on the current state in the game. The red line indicates the optimal move at this point in the game, which is the move that leads to the future state with a heuristic value equal to the maximum utility value assigned to the root node. Figure 3 shows the Python code we constructed to implement the minimax algorithm. The MINIMAXDECISION() function simply calls MAXVALUE() passing it the current board state and which player's turn it is and then returns the optimal move. The MAXVALUE() and MINVALUE() functions recursively call each other as the algorithm simulates the possible future moves and each player's turn. The MINVALUE() function is the same as the MAXVALUE() function except it is searching for a value less than the current minimum utility value while MAXVALUE() is searching for a value greater than the current maximum. In our experiment, the UTILITY() function serves as the static evaluation function to get the heuristic value for a node. We explore this function in greater detail later in this paper.

```
def minimaxDecision(self, board, player, depth):
    tup1 = self.maxValue(board, player, depth-1)
    return tup1[1]      #should return the move that gets to utility value v

def maxValue(self, board, player, depth):
    self.totalNodes+=1
    moves = self.generateMoves(board, player)
    if moves == [] or depth == 0:     #terminal state
        maxTup = (self.utility(board, player), )
        return maxTup
    v = -math.inf       #initialize max to negative infinity so changes if there is any move
    selectedMove = []   #initialize chosen move to be []
    opponent= self.opponent(player)
    for move in moves:  #for every possible move in this state
        testSuc = self.minValue(self.nextBoard(board, player, move), opponent, depth-1)
        if testSuc[0] > v:  #if greater than current max
            v = testSuc[0]  #update max
            selectedMove = move #update chosen move to move associated with new max
    return (v, selectedMove)

def minValue(self, board, player, depth):
    self.totalNodes+=1
    moves = self.generateMoves(board, player)
    if moves == [] or depth == 0:     #terminal state
        minTup = (self.utility(board, player), )
        return minTup
    v = math.inf        #initialize min to infinity so changes if there is any move
    selectedMove = []   #initialize chosen move to be []
    opponent= self.opponent(player)
    for move in moves:  #for every possible move in this state
        testSuc = self.maxValue(self.nextBoard(board, player, move), opponent, depth-1)
        if testSuc[0] < v: #if less than current min
            v = testSuc[0]  #update min
            selectedMove = move #update chosen move to move associated with new min
    return (v, selectedMove)
```

Figure 3: Python code to implement a depth-limiting minimax algorithm.

1.4 Alpha-beta Pruning

Alpha-beta pruning is an additional technique that can be used in combination with the minimax algorithm. An algorithm with alpha-beta pruning will choose the same moves that the basic minimax algorithm would choose at given states in the games, and therefore, should achieve the same statistical chance of winning [2]. However, it uses a pruning method to eliminate branches in the game tree that could not possibly affect the final decision, thereby improving the

computational efficiency of the algorithm significantly. In the Equation 2, we can actually see the potential of using alpha-beta pruning to exclude branches of the game tree that do not need to be evaluated.

$$\begin{aligned}
\text{MINIMAXDECISION}(\text{board}) &= \text{MAX VALUE}(\text{MINVALUE}(3, 12, 8)\,, \text{MINVALUE}(2, x, y)\,, \text{MINVALUE}(14, 5, 2)) \\
&= \text{MAXVALUE}(3, \text{MINVALUE}(2, x, y)\,, 2) \\
&= \text{MAXVALUE}(3, z, 2) \qquad where\ z \leq 2 \\
&= 3
\end{aligned}$$

Equation 2: Demonstrating alpha-beta pruning in action [2].

This equation shows that the minimax-value of the node is independent of the values of x and y so these leaves can be pruned from the tree and never fully evaluated. This method can actually eliminate entire subtrees of a game tree. Figure 4 shows the Python code used to implement the minimax algorithm with alpha-beta pruning, which very closely resembles the basic minimax algorithm shown in Figure 3. Our experiment will examine how much more efficient minimax with alpha-beta pruning can be compared to the basic minimax algorithm.

```
def alphaBetaDecision(self, board, player, depth):
    tup1 = self.maxValue(board, player, -math.inf, math.inf, depth-1)
    return tup1[1] #should return move that gets to utility value v!

def maxValue(self, board, player, alpha, beta, depth):
    self.totalNodes+=1
    moves = self.generateMoves(board, player)
    if moves == [] or depth == 0:
        maxTup = (self.utility(board, player), )
        return maxTup
    v = -math.inf
    selectedMove = []
    opponent= self.opponent(player)
    for move in moves:
        testSuc = self.minValue(self.nextBoard(board, player, move), opponent, alpha, beta, depth-1)
        if testSuc[0] > v:
            v = testSuc[0]
            selectedMove = move
        if v >= beta:
            return (v, selectedMove)
        alpha = max(alpha, v)
    return (v, selectedMove)

def minValue(self, board, player, alpha, beta, depth):
    self.totalNodes+=1
    moves = self.generateMoves(board, player)
    if moves == [] or depth == 0:
        minTup = (self.utility(board, player), )
        return minTup
    v = math.inf
    selectedMove = []
    opponent= self.opponent(player)
    for move in moves:
        testSuc = self.maxValue(self.nextBoard(board, player, move), opponent, alpha, beta, depth-1)
        if testSuc[0] < v:
            v = testSuc[0]
            selectedMove = move
        if v <= alpha:
            return (v, selectedMove)
        beta = min(beta, v)
    return (v, selectedMove)
```

Figure 4: Python code to implement the minimax algorithm with alpha-beta pruning.

2 Previous Work

Although Konane is not a well-known game outside of Hawaii, it is gaining popularity as a tool to study artificial intelligence and combinatorial game theory. Work in the field of artificial intelligence began around the end of World War II; however, the phrase “artificial intelligence” was not coined until 1956 by John McCarthy [2]. Board games are often used to study AI

because, in addition to being complex problems, they are a good representation of human cognition [4]. The first study incorporating AI to play a game occurred in the 1950's when Arthur Samuel wrote a program to play checkers. Samuel also defined characteristics that make a game, or other intellectual activity, a good choice for which "heuristic procedures and learning processes can play a major role" [5]. The game Konane meets all of these requirements as well as to those needed for combinatorial game theory. Konane is not deterministic in nature, meaning there is not a known set of steps to guarantee a win. Furthermore, it is a zero-sum game of perfect information where the game has a definitive end [6]. Zero-sum refers to the outcome of the game. If one player wins, then the other player must lose. Konane is a game of perfect information because, at any state in the game, both players know exactly what moves led to the state, what the current state is, and what moves are possible. Konane has the additional features that no draws are possible and a number moves can be delayed for a turn without losing the opportunity to make the move. In addition to these features, the rules of Konane are easy to understand and implement so Konane is often used as an instrument to teach heuristic algorithms and other AI techniques [7]. Very few published papers explore AI techniques to play the game Konane, but included in these is Darby Thompson's research, which studied training neural networks to play the game [8]. However, many more studies exist that apply AI techniques to similar games. Furthermore, combinatorial game theory studies by Michael D. Ernst and by Alice Chan with Alice Tsai examined the independent sub-games Konane can be divided into whose outcomes can be combined to determine the outcome of the entire game [6]. However, each study only considers certain positions, like in the case of Chan and Tsai who focused on 1xn Konane positions, and concluded more research needed to be done to analyze and document

the various Konane positions and their sums [9]. The studies further validated the use of Konane as a means to study artificial intelligence and combinatorial game theory.

3 Description of Approach

This section describes the methodologies and experimental design used to implement computer agents to play the game Konane.

3.1 Tools Used

For this research, we began with an implementation of the game Konane in Python. The Python code included two main classes, Konane and Player. The Konane class contains functions to generate, validate and make moves, print the current state of the board, and play a specified number of games, as well as other helper functions to gain more information about the game state. The Player class provided a template for the other players we would be designing and declared abstract methods that would need to be implemented by the subclasses. Each new player we created is its own class, but we utilized the inheritance relationship of object-oriented programming languages such as Python [10]. The new player classes we created inherited attributes and methods from the superclasses Konane and Player to decrease redundancy in our code. The inheritance relationship also allows any new player to access the same helper functions provided by the Konane class and the ability to instantiate its own abstract methods declared by the Player class. One abstract method we needed to implement for every player we created was the GETMOVES() method, which is where we provided the strategy or process the player would use to choose the next move. This method was different for each of our players and where we incorporated our AI techniques, like the minimax algorithm, into our players.

In addition, to maintain consistency, all experiments were run on my personal MacBook

Air laptop with a 1.4 GHz Intel Core i5 processor.

3.2 The Players

We created five different Player classes to act as our agents during the game, two of which were computer agents that used AI techniques to chose their move. First, we had a human player that allowed a user to choose a move. We provided a list of potential moves at the current state of the game to give the player all of their options. Then they would be able to input the current row, column location of their piece and the row, column location where they wanted to move their piece. The program would validate that a legal move was entered and perform the necessary actions to reflect the move in the state of the game. For our experiment, I was the human player because of my knowledge of the game and to reduce anomalies in our results. We also had a random player that simply chose a random move from the list of potential valid moves. We used the Python random library to generate a pseudo-random number in the range zero to the length of the possible moves list. The number that was generated was used as an index into the list of possible moves and the move at that index became the chosen move. Our next player was arguably our most worthy opponent for our computer agents because we incorporated some strategy into choosing its move. Therefore, this player was called the strategy player. Four elements were incorporated into how this player chose a move. First, the strategy player never moves a corner piece unless there is no other option for a move. Empty spaces on the board expand from center of the game board or an opposite corner, where the games pieces were removed from the board at the beginning of the game. Corner pieces are often some of the last pieces moved because empty spaces reach these areas of the board later in the game. These positions provide safety for a player's piece while also allowing them to reserve a move from turn to turn assuming an adjacent opponent's piece has not been moved. Second, this player

moves a piece that is in immediate danger of being jumped by an opponent's piece, and third, tries to not move pieces not in immediate danger of being jumped. Lastly, this player attempts to prepare for future moves by analyzing how many moves are possible for their next turn for each of the current possible moves, without taking into consideration their opponent's move. For our first AI agent, we created a minimax player that used the depth limiting minimax algorithm to choose a move. For each move, this player called the MINIMAXDECISION() function shown in Figure 3 which then recursively called the MAXVALUE() and MINVALUE() functions until a terminal state was found, or the depth limit is reached. The move chosen is the move that led to the returned utility value. Our other AI agent was the alpha-beta player. To choose a move, this player also utilized the depth limiting minimax algorithm but included the alpha-beta pruning technique. For each move, this player called the ALPHABETADECISION() function shown in Figure 4, which behaves similarly to the MINIMAXDECISION() function.

3.3 Experimental Design

We designed two separate experiments in order to fully understand the effects of using AI techniques to play the game Konane. However, some aspects of the two experiments remained constant. We decided to use an 8x8 size game board for Konane as shown in Figure 1. We wanted to balance a large enough search space with a reasonably sized game that would not be too computationally expensive. Also, Joel Gyllenskog stated in his study of Konane that "strategies for winning are independent of the board's size" so we assume our findings could be replicated and scaled with larger board sizes [7].

Our first experiment focused solely on how effective our AI agents were at playing the game Konane. We set up a round robin tournament so that each of our AI agents played the other four agents, including the other AI agent. In this part of the experiment, we limited our two AI

players to a look ahead depth of five instead of searching the entire depth of the tree to find a terminal state. Setting a limit for the depth helps handle the complexity of the algorithm when the game has a high branching factor, like Konane which has an average branching factor of ten [8]. The two AI agents played the random player, the strategy player and each other one hundred times alternating which player went first every game. The human player only played each AI agent twenty-five times since the game waited for a response from a user while the other games were automated and able to run without supervision. We recorded what percentage of games our AI agents won as a measure of their success.

Our second experiment studied the difference in efficiency of each AI agent. We had each AI agent use varying depths and play the strategy player twenty-five times at each depth again alternating which player went first. We observed changes in the average time it took the AI agents to make a move as well as how many total nodes the agents explored while playing each game at a depth of two, four and six.

3.4 Utility Function

Both of our AI agents utilize a static evaluation function to get the heuristic value of various states in the game. The UTILITY() function is the helper function we created to calculate these values which are returned to the MAXVALUE() and MINVALUE() functions of the minimax algorithm as shown in Figures 3 and 4. In our implementation, THE UTILITY() function takes into account the size of the game board and how many of a player's game pieces are left as an indication of how quickly the player was able to reach a terminal state in the game. When a player wins the game, the size of the game board squared is added in the calculations but if the player loses, this number is subtracted. Also, if we enter the UTILITY() function but are not in a terminal state, we know the algorithm reached the depth we set for the search, and we are simply

looking for an approximation of the best move at a given state in the game. In this case, we still look at board size but also look at how many potential moves a player has from this state, as an indication of how good a position is for a player. We needed to ensure the value calculated when a node was not a terminal node could never be less than or greater than a terminal state so we used the implementation of the UTILITY() function shown in Figure 5.

```
def utility(self, board, player):
    """
    Calculates the utility value for a terminal state in the game
    Takes into consideration how many of your own pieces left,
        which correlates to time took to win, and whether or not won
    """
    if self.generateMoves(board, player) == []: #player looses
        value = (self.countSymbol(board, player) * 3) - (self.size ** 2)
    elif self.generateMoves(board, self.opponent(player)) == []: #opponent looses
        value = (self.countSymbol(board, player) * 3) + (self.size ** 2)
    else: #game not over, so look at how many moves to choose from
        value = (len(self.generateMoves(board, player)) * 3) + (self.size * 2)
    return value
```

Figure 5: Python code for UTILITY() function.

4 Results

Overall, our results for the two experiments were what we would have expected. For the first part of the experiment, we can see the results of what percentage of games were won in Figure 6. Both AI agents beat the human player unfettered and beat the random player and strategy player close to ninety percent of the time. This is reasonable because, with a limited depth of five, these agents should be effective but not necessarily one hundred percent effective. In addition, our human player was not a master at Konane so these results may be slightly skewed. Furthermore, we observed that the two AI agents were equally as effective against the other players and each other. This reinforces the idea that both agents use the same basic algorithm, with added features for computational efficiency, so they can achieve the same level of success. In particular, when

they played each other, it was typically the player who went second who won the game. These two computer agents, only when set at the same depth, actually chose the same moves to get to the same end game. We witnessed a similar phenomenon when we first began running tests with an agent that utilized the minimax algorithm on a 4x4 size game board playing one game at a time. Upon further investigation, we confirmed that a player who goes second in a 4x4 game can limit the first player to only one move a turn and can win every time [9]. This shows the importance of alternating which player went first each game in our implementation.

	Minimax Agent	**Alpha-beta Agent**
Wins against Human Agent:	100%	100%
Wins against Random Agent:	90%	91%
Wins against Strategy Agent:	89%	89%
Wins against other AI Agent:	~50%	~50%

Figure 6: Results from part one of the experiment showing percentage of wins.

For the second part of the experiment, we looked at the efficiency of the AI agents compared to each other. The agent that employed the minimax algorithm with alpha-beta pruning was able to win in a drastically shorter amount of time by exploring significantly fewer nodes every time. Figure 7a shows the numerical results of both measures in a table while Figure 7b and 7c use the data from Figure 7a to show the results visually separated by measure while emphasizing the difference between the two agents. The two graphs each map a measure as a function of the depth of the search tree. We used a logarithmic scale along the y-axes in both graphs. This implies choosing a move grows exponentially as the depth of the search increases. Likewise, the number of nodes explored increases exponentially as more nodes as considered at greater depths. In order to see the correlation between the success rate and search depth, we ran

another test using the minimax with alpha-beta pruning agent at various depths against the random player and can see the percentage of wins increase as search depth increases in Figure 8.

	Minimax Agent			Alpha-beta Agent		
Tree Depth:	**2**	**4**	**6**	**2**	**4**	**6**
Total Number of Nodes Explored:	189	50,093	4,245,710	178	11,585	202,735
Average Time to Make a Move (ms):	7.93	2,683.9	140,721.0	6.45	430.78	6,344.4

Figure 7a: Table of results from part 2 of the experiment showing the total nodes explored and average time to make a move for the two AI agents at varying depth limits.

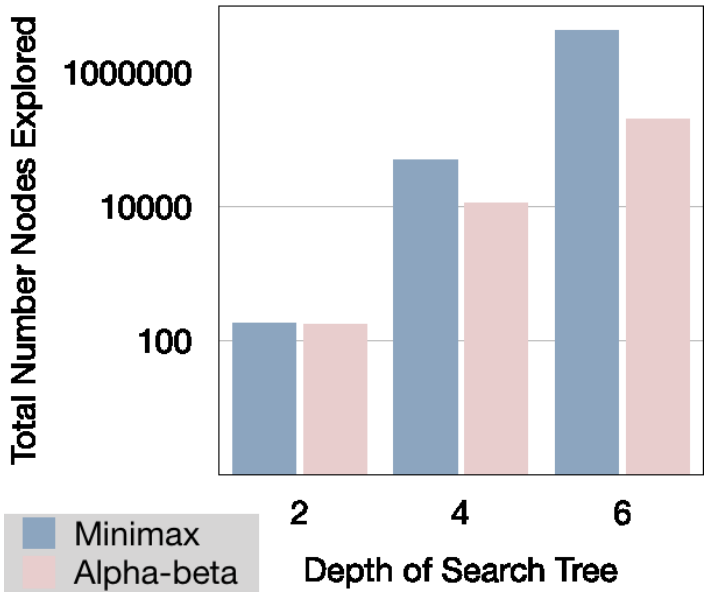

Figure 7b: Graph of results from the experiment part two – total number of nodes explored as a function of the depth of the search tree.

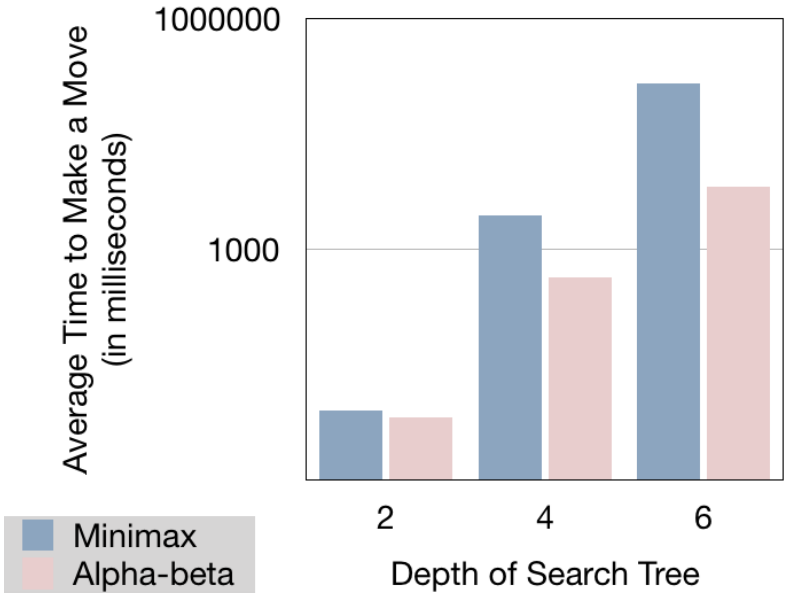

Figure 7c: Graph of results from the experiment part two – average time to make a move in milliseconds as s function of the depth of the search tree.

	Alpha-beta Agent					
Tree Depth:	**2**	**3**	**4**	**5**	**6**	**7**
Wins against Random:	34%	56%	77%	91%	93%	94%

Figure 8: Table of results from showing the success rate of the alpha-beta running agent at various depths against the random player.

5 Future Work

In this project, we were able to successfully implement effective computer agents using artificial intelligence algorithms to play the game Konane. However, there remain many areas to be explored around the game Konane. Continuing to study the positions of the game and improving

the computer agents are among the next steps. Addressing weaknesses in the static evaluation function can enhance the minimax algorithm. One way to accomplish this is to randomize the move selected when the utility values of two nodes are equal. This would eliminate the potential of being beat repeatedly by a player exploiting the fact that the algorithm will choose the same move given the same state. With this slight change, if a path to defeat the AI agent is achieved, it will most likely not be replicated because of the statistical element added to the choice. When alpha-beta pruning is used, the same effect can be achieved by randomizing the order in which the list of potential moves is analyzed. This can lead to different moves being selected as the alpha and beta values and therefore different branches being evaluated or not evaluated [6]. Furthermore, a greater understanding of Konane positions as it relates to solving smaller sub-games would allow us to choose better branches to evaluate overall and potentially be able to evaluate better heuristic values for the nodes that are not terminal states. Optimizing the order in which moves are evaluated with alpha-beta pruning leaves room for studies on how Konane experts choose which moves to evaluate during the game [11].

In addition to improving weakness in the static evaluation function, we can use this function to our advantage to implement various strategies that put emphasis on different aspects of the game. With more time, I would have liked to do more testing around changing the coefficients and variables of the game involved in the static evaluation function. By simply adjusting the UTILITY() function, we could assign more weight to different elements in the decision to choose a move. By doing this, we can create players with different intermediate goals. Furthermore, since Konane has such unique stages of the game and a limited number of moves near the beginning and end of the game, different strategies can be employed in these two smaller subspaces. It is even possible to have a set of beginning moves that do not require

evaluation because we know in advance the best move up to a point or until the opponent varies off the set path [11]. More work in the area of combinatorial game theory to solve subgames could also lead to better heuristic evaluations of terminal nodes or even employing a predetermined set of moves to win a game in the end stages if it got to a subgame that has been solved. Although players that include the improvements above provide more flexibility and variation in the gameplay, the creative potential of the human mind can never be translated to a computer agent.

Lastly, areas of artificial intelligence that train and learn can develop even more sophisticated computer agents. Thompson's paper on neural networks for the game Konane was a great start but more work can be done [8]. In order to evaluate the performance of any these various computer agents, a comparison must always be made against a known existing alternative, such as the minimax algorithm or an expert human player.

6 Conclusions

During our experiments, we observed the effectiveness of computer agents using the minimax algorithm to choose a move in the game Konane. We were able to measure the success of computer agents using artificial intelligence by examining the percentage of wins against our other three agents. The agents that utilized the minimax algorithm at a limited search depth of five won about ninety percent of the time. The minimax agent and minimax with alpha-beta pruning agent achieved the same success rates and, in fact, chose the same move given the same state of the game. This reinforces the concepts that they rely on the same algorithm and the need for improvements in the algorithm to add statistical variation. The second part of the experiment confirmed the minimax with alpha-beta pruning agent was significantly more efficient when choosing a move in terms of total nodes explored and average time to make a move than the

basic minimax agent. Furthermore, we discovered the relationship between the total number of nodes explored during the search is exponential in terms of the depth of the search tree, and the same relationship exists between the average time to make a move and the depth of the search tree. Finally, we concluded that increasing the depth of the search does lead to greater success. The minimax algorithm provides a sufficient strategy for the game Konane that can be used as a comparison in future studies on the effectiveness of other computer agents.

Acknowledgments

Foremost, I would like to express my sincere gratitude to my advisor Dr. Michael Scherger for his guidance, patience, and constant support throughout the duration of this project. I would also like to thank my thesis committee, Dr. Antonio Sanchez and Dr. Wendy Williams, for their encouragement, insightful comments, and hard questions as well as the rest of the Department of Computer Science for their support throughout my years in the Computer Science program. Lastly, I would like to thank Lisa Meeden at Swarthmore College for sharing the Python code implementation for the game Konane and the corresponding helper functions.

References

1. Hiroa, T. R. (1957). *Arts and Crafts of Hawaii*. Honolulu, HI: Bishop Museum Press.
2. Russell, S. & Norvig, P. (2003). *Artificial Intelligence: A Modern Approach (2nd ed.)*. Upper Saddle River, NJ: Prentice Hall Press.
3. Poole, D., Mackworth, A., & Goebel, R. (1997). *Computational Intelligence: A Logical Approach*. Oxford, UK: Oxford University Press.
4. Rasskin-Gutman, D. (2009). *Chess Metaphors: Artificial Intelligence and the Human Mind*. (D. Klosky, Trans.) MIT Press. (Original work published 2005)
5. Samuel, A. L. (2000). "Some Studies in Machine Learning Using the Game of Checkers." *IBM Journal of Research and Development, 44.* Retrieved from http://library.tcu.edu.ezproxy.tcu.edu/PURL/EZproxy_link.asp?http://search.proquest.com.ezproxy.tcu.edu/docview/220681210?accountid=7090
6. Ernst, M. D. (1995). "Playing Konane Mathematically: A Combinatorial Game-Theoretic Analysis." *UMAP Journal, 16.* Retrieved from https://homes.cs.washington.edu/~mernst/pubs/konane-tr9524.pdf
7. Gyllenskog, J. H. (1976). "Konane as a vehicle for teaching AI. Intelligence." *ACM SIGART Bulletin, 56.* Retrieved from http://delivery.acm.org.ezproxy.tcu.edu/10.1145/1050000/1045261/p5-gyllenskog.pdf?ip=138.237.48.235&id=1045261&acc=ACTIVE%20SERVICE&key=F82E6B88364EF649%2E8CB32E69A49AB401%2E4D4702B0C3E38B35%2E4D4702B0C3E38B35&__acm__=1525709876_dd8fa9bd02ee0629c0d5ec906275c4fa
8. Thompson, D. (2018). "Teaching a Neural Network to Play Konane." Retrieved from https://cs.brynmawr.edu/Theses/Thompson.pdf

9. Chan, A. & Tsai, A. (2002) "1xn Konane: A Summary of Results." *MSRI Publications, 42.* Retrieved from http://library.msri.org/books/Book42/files/chan.pdf

10. https://www.python.org

11. Shannon, C. E. (1950). "XXII. Programming a Computer for Playing Chess," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science, 41.* Retrieved from https://vision.unipv.it/IA1/aa2009-2010/ProgrammingaComputerforPlayingChess.pdf