The Alpha-Beta algorithm in Zero-Sum Board Games

Nabil Mannan
170238818
Matthew Huntbach
MSc Computer Science

***Abstract*— The research contained in this paper focuses on what the alpha-beta algorithm is and how it can be used in Artificial Intelligence (AI) to determine strategic moves an AI can deploy against a human. The findings of this report can also help supplement and establish a steppingstone for AIs to think for themselves and operate without human interference. Although the algorithm has been seen before, comparisons of different variations in a game such as tic-tac-toe or connect-4 has not been widely investigated. Hence, this project aims to implement the pruning technique in zero-sum board games (i.e., tic-tac-toe and connect-4) and determine the best variations for any difficulty level. It should be noted that alpha-beta algorithm and alpha-beta pruning are regarded as the same in this paper.**

***Keywords*— alpha-beta algorithm, pruning, Artificial Intelligence (AI), zero-sum game**

I. Introduction

In 1920 a famous mathematician John Von Neumann established the central problem of game theory. Assume there are players 1...m, who are playing a game G. Which moves should player n make, to achieve the best maximum payoff? (Burguillo C. J. 2018). Since then, there have been a number of algorithms created to tackle this problem with the most common and significantly successful, being the minimax algorithm. As a result, more academics have explored search algorithms and, "AI has developed superhuman intelligence" (Kang et al., 2019). A perfect example of this is Deep Blue, a chess-playing system designed by IBM, which defeated world champion Garry Kasparov in 1997. Also, Google's DeepMind company has developed a highly intelligent AI player, Alphazero, (Kang et al., 2019) that has mastered games such as chess, shogi and go. This paper will discuss and implement the alpha-beta algorithm in two, 2-player board games. As well as also attempting to discover an optimal variation of the algorithm to achieve even better results.

However, before considering the alpha-beta algorithm one needs to understand what the minimax algorithm is. The alpha-beta algorithm is an extension of the minimax algorithm, like several others. The minimax algorithm first proven by game-theorist mathematician, John von Neumann in 1928, uses a recursive approach in a range of fields from decision theory, game theory, AI and more. The purpose of the algorithm is to provide a player with an optimal move assuming that every other player is also playing optimally. Simply, its purpose is to minimise the potential loss for a maximum loss situation. The algorithm is most commonly used in two player, zero-sum games (relationships or in this case, games between two entities where if one person gains, the other person loses). Board games are a simple example of zero-sum games, where a player can be at an advantage if they make the optimal move, causing their opponent to have a disadvantage. These techniques can only be applied to games with perfect information, where it is possible to see and predict an opponent's moves (Hacker earth, 2017), i.e., checkers, and tic-tac-toe.

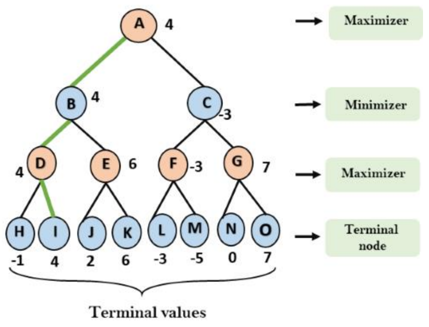


Figure 1. Visualisation of the minimax algorithm on a game tree

II. Background

This section describes in detail how both the minimax and alpha-beta algorithms function. Where minimax is a necessary prerequisite for alpha-beta pruning.

A. *Minimax algorithm*

The minimax algorithm is used in AI for decision making. It is mostly applied in turn-based two player games such as connect-4, chess, etc. The algorithm "aims to find the optimal move for a player, assuming that the opponent also plays optimally" (Kang et al., 2019). The minimax algorithm begins with two players, MIN, who wants to achieve the minimum possible score and MAX, who wants the maximum possible score. This can be visualized on a game tree, where the root of the tree is MAX, and its immediate successors are MIN. Followed again by MIN's successors which are MAX (Igor and Judea, 1982). Minimax applies the depth first search (DFS) algorithm which traverses the whole tree from root to terminal, exploring every branch, before backtracking. Although (Eka Putri et al., 2011) implements DFS to find the longest path, the minimax algorithm searches from its terminal node to its root, which is always the longest path.
We will observe figure 1 to help understand how the algorithm works

- Step 1 generates the game tree required for the chosen game i.e., figure 1. The figure shows the current position A, through to the final position where the game ends at the terminal node that contains the scores for each move.
- Step 2 works backwards from the terminal node. In the case above it is the MAX players turn, so

for nodes D, E, F, and G we will choose the max number from their pairs, i.e., node D has pair H and I. Hence, the utility for D is 4 since it is greater than -1, evaluating max {6,2} for node E returns 6 and so on for the rest of the nodes during that turn.

- Step 3 moves onto the next player who is MIN with nodes B and C. The objective of player MIN is to obtain the lowest possible utility from the previous MAX players turn. For example, in figure 1, player MIN wants min {4,6} for node B since previously player MAX chose 4 for node B, and 6 for node E.
- The above processes are repeated, calculating the utility for every layer until you reach the root of the tree. Where the final utility will be obtained by player MAX choosing the greatest value which in figure 1 is given as 4.

For the tree above the best move/decision for MAX would be to choose the highlighted (green) path IDBA where the greatest utility they can obtain is 4. "This move is called the minimax decision as it maximizes the utility following the assumption that the opponent is also playing optimally to minimize it" (Hacker earth, 2017). However, as (Russell and Norvig, 2010) mention, "The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree." This difficulty affects games such as chess that have many moves for the algorithm to predict (i.e., different pieces have different movement patterns) and therefore creates a tree that contains many branches and nodes. As a result of this, the method of alpha-beta pruning was discovered to reduce search time whilst still providing the same results.

B. *Alpha-beta algorithm*

The alpha-beta algorithm or also known as alpha-beta pruning, is an optimised and enhanced version of the minimax algorithm. As we saw previously from minimax, the algorithm searches and evaluates every node in the tree (DFS) and considering there are trees that may become extremely large , this search process becomes exponentially long. Therefore, (Igor and Judea, 1982) imply that the algorithm is an effective solution.by stating that, "the alpha-beta algorithm… significantly slows the exponential growth of game-tree searching using a pruning technique". Since we cannot just remove nodes (the moves each player can take) we can instead skip over them, as in we do not need to calculate the minimum, or maximum for every move. Which still results in the same minimax value. The element of not evaluating every node in the tree is called pruning and is the reason the alpha-beta algorithm is so fast.

A key difference between the algorithms is the addition of two parameters alpha and beta. Where alpha represents the best move (which is the greatest value) at any place along the path for player MAX. And beta represents the best move (which is the lowest value) at any point for player MIN. Alpha always begins at -∞ and beta at +∞. This is to ensure that there is always a better choice to choose when evaluating each node. The condition required to prune a branch or even an entire sub-tree is that alpha >= beta and can be applied at any depth of the tree (Russel and Norvig, 2010). To understand how the alpha-beta algorithm works, study figure 2.

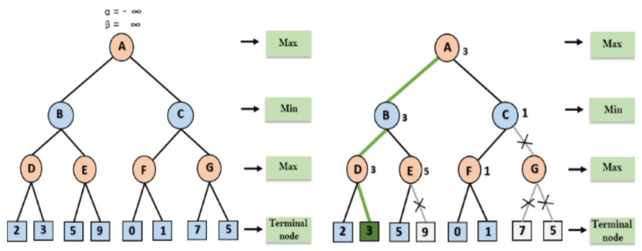


Figure 2. Alpha-beta algorithm before and after pruning

- Say we begin with player MAX as before, the algorithm will traverse the tree along with the values of alpha and beta, until it reaches the parent node (i.e., D) of the first pair of terminal nodes. As you reach node D, which is the turn of player MAX, the value of alpha (since it is player MAX) is updated to the result of max {2,3} which is 3. The algorithm will then backtrack to node B where beta is updated to min {∞,3} which is 3, and alpha will also return to -∞ (at B it would be player MIN's turn). The values of alpha and beta then traverse down to the next node E, where the process repeats.

- At node E it is player MAX's turn and hence only alpha is influenced. Looking at the first child of E (i.e., 5), we evaluate max{-∞,5}, which is 5. So, we update our parameters to alpha=5 and beta=3 (beta from previous step). Then, looking at the condition to prune where alpha>=beta, we can see that indeed 5>=3 and therefore we can prune the remaining child of E resulting in the value of 5 at node E (We did not consider the remaining child, 9, as it would have made no difference).

- Now as it is player MIN's turn, the value at B remains 3 since it is the smaller value of min {3,5}. This will then backtrack to node A where the new parameters become alpha=3, and beta=∞ (since again it is the turn of player MAX).

- These values of alpha and beta will then traverse down the tree for the next branch. This process will continue until all branches or sub-trees have been either evaluated or pruned, resulting in the same answer given by the minimax algorithm.

III. Alternative Algorithms and Related Work

The following will discuss alternative algorithms and methods to alpha-beta, that are currently or have been used in the past. Presenting their advantages/disadvantages over alpha-beta and how they compare with different depths.

1) *Heuristics for Minimax algorithm*

There are opportunities for the base minimax algorithm itself to be improved for its efficiency. In the journal by (Kang et al., 2019), they compare two different heuristics that improve the functionality of the minimax algorithm. A heuristic is used to find quick solutions to slow classical methods. (Kang et al., 2019) established that there are two main factors that contribute to the effectiveness of the heuristic function, i.e., depth searching and number of features. Depth searching is the chosen layer of the game tree the algorithm will search up to (generally more time is needed to search further layers). In the context of connect-4, the number of features represent the different game states you can be in when playing. For example, "Feature 4 is the occasion when a chessman is not adjacently connected to any other same chessman" (Kang et al., 2019). A chessman is regarded as the piece/colour played with. Only using feature 1 and 2 resulted in a winning percentage of 0.54 whereas using all features resulted in 0.80. It is then concluded that the increase in depth of searching returns a heuristic with better functionality. Furthermore, an optimal heuristic function is achieved by combining all features. Unfortunately, the scope of this research ends at only one game, connect-4. Therefore, it cannot be concluded that for more complicated games such as checkers, or Othello, heuristics will prove to be faster and more effective. (Buro, 2002), however uses another heuristic, Multi-ProbCut (MPC) to prune "at different search depths, uses game-stage dependent cut thresholds, and conducts shallow check searches". MPC beats regular ProbCut (PC) 72% out of 140 games, where PC is already superior to alpha-beta searching. Therefore, (Buro, 2002) concludes that "MPC significantly outperforms" alpha-beta pruning and that the alpha-beta algorithm is slower due to it analysing irrelevant branches.

2) *MTD(f)*

The implementation of algorithms to build supercomputers with the ability to predict patterns is a common field of interest and research. MTD(f) (Memory-enhanced test driver) is an example of such an algorithm. Aske Plaat, a Dutch AI scientist, discovered a new combination of the alpha–beta algorithm with transposition tables, "to overcome the drawbacks of the original algorithm and develop a new variant called MTD(f)" (Russel & Norvig, 2010), that has been accepted by various top programmers. The MTD(f) algorithm searches a game tree and stores the nodes in memory giving it the ability to re-search game trees at a fast and efficient rate. Repeated calls on MTD(f) can be used to home in on the correct minimax value (Aske et al., 2014). In the paper on "the analysis of the alpha-beta pruning and (MTD(f))" by (Lukas et al., 2017), it compares the two algorithms and determines which one has the fastest execution, and evaluation speeds on the board game connect-4. (Lukas et al., 2017) concludes that MTD(f) is on average faster and performs better in every category. One reason for this is due to the MTD(f) algorithm evaluating fewer leaf nodes than the alpha-beta algorithm. However, we cannot conclude that MTD(f) is overall better than alpha-beta pruning since the experiment was only assessed on one game (i.e., connect-4), and performance was recorded only at certain depths (node layer) of 5-10.

3) *SSS* algorithm*

The SSS* (State Space Search) algorithm uses state space searching (i.e., different states/instances of the game are considered), to perform parallel tree traversals, hence the name. It has been proven that SSS* is a valid minimax algorithm and that during its run it never investigates nodes that the alpha-beta algorithm has already pruned (Stockman, 1978). Simply, SSS* builds a tree imitating best-first process by searching nodes in order from most promising and then the least, whereas alpha-beta uses depth-first search. When both SSS* and alpha-beta are given the same perfectly ordered game tree, they visit the same nodes, however on average SSS* would evaluate significantly fewer nodes. To understand how the SSS* algorithm functions would require a high-level description as well as understanding of many other works in Stockman's article found here (Stockman, 1978). Consequently, there are many disadvantages to this algorithm given in the (Technical report by Platt et al.). The algorithm is too complex and difficult to understand, making it challenging to implement in different applications. The necessity of having a large memory makes it impractical for the real world. As a result of these impracticalities, it has become a deterrent for academics to use in practice and hence alpha-beta pruning is still widely preferred over it.

IV. METHODOLOGY

Here the methodology is split into two sub-components, tic-tac-toe, and connect-4. Which is then further divided into user interface (UI), standard AI build and alpha-beta enhanced build. Where the UI component aims to explain how the game runs and what the different messages mean. Standard build and alpha-beta build explain the code structure used in the experiment. Followed by any variations to the code to make the AI harder or easier to beat.

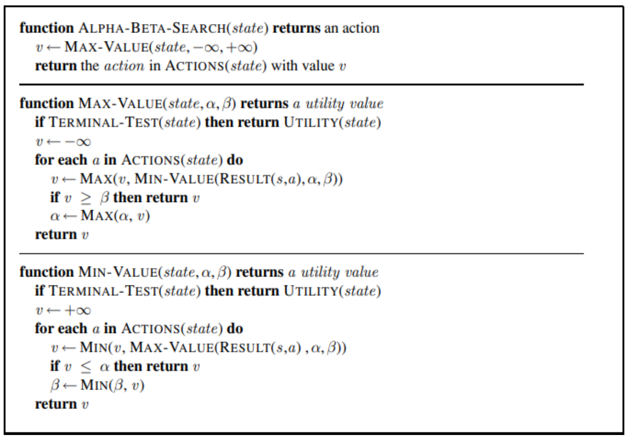
A. *Tic-Tac-Toe*

1) *User Interface explanation*

Running and playing the game is quite simple as there are instructions printed throughout the game that help you during each step. The first statement displayed after starting the game will welcome you to tic-tac-toe against an AI. Followed by a question asking the player if they want to move first or not. The player then needs to type either 'Y' for yes to go first, or 'N' for no to go second. Selecting anything but 'Y' or 'N' will return a message saying 'Try again' asking for the input to be a string. Once a valid string is given, the game will then ask the player to choose a number between 1 and 9. Where the numbers 1 to 9 represent the keys on a number pad i.e., on a laptop or calculator. To make selection easier, the game board is displayed with the integers of all positions. After a valid number is chosen the game will display the tic-tac-toe board with the players board piece, 'X', in the position they chose. Note that the player is always 'X' and AI is always 'O'. Immediately after the player has made their move, the game will display a message 'Next turn' to indicate that the board has been updated with the AI's move. The game may show a message saying that the position has been taken, multiple times as the AI is randomly selecting a position. This position could already have been selected in previous rounds by either player or the AI. The process of selecting a move will continue until either all positions are filled resulting in a draw or if the player wins or the AI wins.

2) *Standard AI build*

To create tic-tac-toe the use of several definitions and functions is required for the game to run successfully. The first step was by starting with the building of a board to operate on using a 3x3 array. I then filled the array with empty variables ‘FREE’ to indicate that the position is not being used. Next, I moved onto the function that adds and updates the board after every turn, using the board pieces. Thus, making use of a key and value pair I first created the board pieces, ‘X’, ‘O’, and ‘FREE’, for the player, AI, and empty positions respectively. Using nested for loops, the 'game board' function iterates over the positions of the board and replaces the values of ‘FREE’ with the values of the keys, which are the board pieces, after each turn. For example, an empty string will be replaced by an ‘X’ or ‘O’.
I also needed the game to understand what a win was. To do this, I created another array that contained all possible winning combinations, i.e., 3 in row, column or diagonal. Hence, the player or AI would be classed as a winner if they matched any of the combinations during the game.
Also, after the game has ended, I wanted a function to display the results indicating who won. Therefore, using the previous function ‘winner’, that determines who is the winner, I used for statements to print ‘You have won!’ if the player wins, ‘Unlucky, you lose!’ if the AI wins, and ‘It’s a draw’ if the result is a tie.
The program also needed to understand that if the board is full of board pieces the game is finished. As well as determining where the empty spaces to be filled are positioned. The prior is simpler as the function only needed to know that if the number of free spaces is equal to zero then the game has ended otherwise continue playing.
The ‘free spaces’ function returns a list with the coordinates of any and every free space i.e., position is filled with FREE and not a board piece. What this means is, say that a player has made a move, for example in the middle of the board. The function would return every coordinate excluding the middle coordinate, [1,1]. And therefore, the game would not allow the next player to choose this position as it is now not in the list of coordinates to choose from. This prevents the problem of the AI selecting a position the player has already selected. To actually create the coordinates, I used the enumerate function to record the index of both the row and column of each position on the board. Then appended the indices to an empty list if the value of that position was equal to FREE.
The most important part of designing tic-tac-toe is getting the players and AI’s moves working correctly. To do this I decided to start with a while loop to ensure that the taking turn process repeats itself until game over. Then I needed the function input, that would allow a player to interact with the game by selecting a move of their choosing. Next, using an if statement I restricted the choice of move between 1 and 9 as well as preventing the player and AI from being allowed to choose an already selected position i.e., if the position is not in free spaces, it cannot be selected. Once all the above conditions are satisfied the else statement comes into effect where the board piece is permitted to move. This required a small dictionary with keys from 1-9 and their values equivalent to the coordinates of the numbers on a number pad, i.e., 1 is top left corner so [0,0] and 9 is bottom right corner so [2,2]. The rest of this function updates the



```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 3. Pseudo-code for alpha-beta algorithm

board with the players move if it is legal. For the AIs move, instead of allowing an external input all it requires is a random integer between 1 and 9. And would continue to randomise until a legal move is made and permitted to move.
Lastly, I combined all the aforementioned functions and definitions to create the final game. Similarly, to before I opened up with a while loop and used the input function to allow the player to decide whether to go first or second. Using the try function ensures that the player has to choose a valid option to move on. Then if the value of input is ‘Y’, the player moves first else it is the turn of the AI. The next loop ensures that the game continues as long as the game has not ended, or the board is full. This needed the function ‘full board’ where if the length of free spaces was equal to 0 the game would end, since there are no more spaces to move. I also needed another function ‘end game’ that returns the winner of the game which is determined from the previous function ‘winner’. So, as long as these conditions are not met in the second while loop, the player and AI will continue alternating, taking turns.

3) *Alpha-beta enhanced build*

Implementing the functions described in part C below into the function that moves the AI just needed a variable called result. This is the result of evaluating the alpha-beta function with parameters board, the depth (number of free spaces on the board), alpha (-infinity), beta (infinity) and the player which in this case is the AI.

B. Connect-4

1) *User Interface*

As soon as connect-4 is initiated you are welcomed with a similar welcoming message to tic-tac-toe, followed by the board that will be played with, along with the first set of instructions asking whether the player wants to move first or not. The player is then required to type the string they would like to play either ‘Y’ or ‘N’. Selecting to play first will then immediately trigger the game to ask what move to make first. Specifically, the game will ask the player to choose an integer from a range of 0 to 6. In order to make gameplay smoother there is an index above the game board indicating where each position is located, to save players time in working out where to place their piece. After the player has chosen a valid integer, the board will update with the number ‘1’ that indicates their piece followed by the message ‘Computer calculating next move…’, where the AI will then place their piece ‘2’. The board will again update with the AI’s piece and then ask the player to choose the next move. This process is then repeated until either the player or AI matches four of their pieces or until there are no more moves to be made. Where the game will end in a draw.

2) Standard AI build

When creating the connect-4 board game I reused a lot of code from tic-tac-toe which made it slightly smoother. I first defined the function that builds the board, using NumPy to build a 6x7 array filled with zeros.

I then needed a function that actually places the board pieces, which takes the row and column indices as its arguments. This process could either be a function or written out in full, which are both single lines of code. I decided to leave it as a strand of code rather than create a new function.

The next step requires a function ‘has space’, that determines if the moves are legal or when the column is full. Similar to the previous function I used index slicing where the function returns true if the last row and chosen column is empty else false.

I then needed a way to determine at what index is the next empty position, i.e., the index of the row after the players or AI’s pieces are stacked. This lead to the creation of the function ‘next free row’. This returns the index of the row where the next empty space is available for the chosen column. To then make this look like an actual connect-4 game I then used the NumPy flip function which reflects the array on a chosen axis i.e., vertically. Now the pieces will look like they are dropping and being stacked.

After completing the conditions for moving the pieces I then moved onto determining how the winner is found. The first step was to find a horizontal win i.e., four same board pieces are stacked alongside each other. Hence, again using indices, the condition to win horizontally is that there has to be four matching pieces. For example, the board piece at position [1,1] has to be equal to [1,2], [1,3] and [1,4]. A similar argument is used for a vertical win, however instead of looking at the row it focuses on the columns. For the diagonal wins, there needs to be two separate conditions to check, an upward sloping and downward sloping match. Where the slicing is incremented by one for both row and column for the upward sloping diagonal. Whereas for the downward sloping diagonal the row is decremented by one.

Defining the players move is almost identical to tic-tac-toe with a few small tweaks. First opening up with a while loop and defining variable ‘col’ which is the integer the player wants to move, as well as restricting their input between 0 and 6. Then using the previous ‘is valid’ function to check whether the player can move to that position. If these conditions are fulfilled, then the integer is run through the ‘next free row’ function to calculate what row the board piece will fall to. Then finally printing the board with the updated move. For the AI’s move, the input takes a randomly generated integer between 0 and 6 and therefore does not need any of the restrictions the player has.

Similar to tic-tac-toe I reused the ‘who won’ function that displays the result of the game. And also, the ‘finished’ function that ends the game if there is a winner.

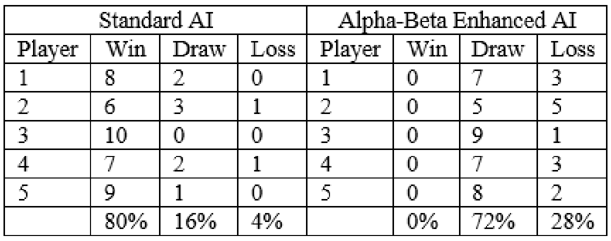
Finally, the last function ‘game’ describes the process of the player choosing to move first or second, as well as taking turns between player and AI which is exactly the same as tic-tac-toe. A while loop is used to determine who starts first by asking the player to choose either ‘Y’ or ‘N’. The players then continue to alternate until the game has ended followed by a statement declaring the outcome.

3) Alpha-beta enhanced build

Again, similar to tic-tac-toe, to implement part C, I created a variable, result, which is equal to the evaluation of the alpha-beta algorithm. The parameters used are the board, depth of 4, alpha, beta and -1 (AI). The ‘next free row’ function then evaluates the result variable to give the best move out of the available positions. Finally, the board is then updated with the piece and the new board is printed.

C. Alpha-beta (both games)

Applying the alpha-beta algorithm requires defining the function and then implementing it into the ‘AI’s move’ to use it. The first step when creating the alpha-beta algorithm is to return the score when the depth of search is at the first layer of the tree, i.e., at root node equivalent to depth=0. This means that the game has already ended since there are no moves to choose from, meaning nothing is being searched. Now if the depth is not zero, then the next step is, using a for loop, the function is to scan every free space on the board and apply the alpha-beta algorithm to get an evaluation. If the maximising player is the human, then the evaluation will be compared to alpha (which is always -infinity). The greater of the two, alpha and the evaluation, becomes the new alpha. On the other hand, if the maximising player is the AI, the process is similar, however this time compares the evaluation with beta (which is +infinity). Also, the new beta value becomes the smaller between the two, evaluation and beta. After determining the new values of alpha and beta, compare the two. If alpha is greater than or equal to beta the loop should end. I also decided to use a time module to record the evaluation



Standard AI				Alpha-Beta Enhanced AI			
Player	Win	Draw	Loss	Player	Win	Draw	Loss
1	8	2	0	1	0	7	3
2	6	3	1	2	0	5	5
3	10	0	0	3	0	9	1
4	7	2	1	4	0	7	3
5	9	1	0	5	0	8	2
	80%	16%	4%		0%	72%	28%

Figure 4. Results from 5 different players against AI with and without alpha-beta pruning

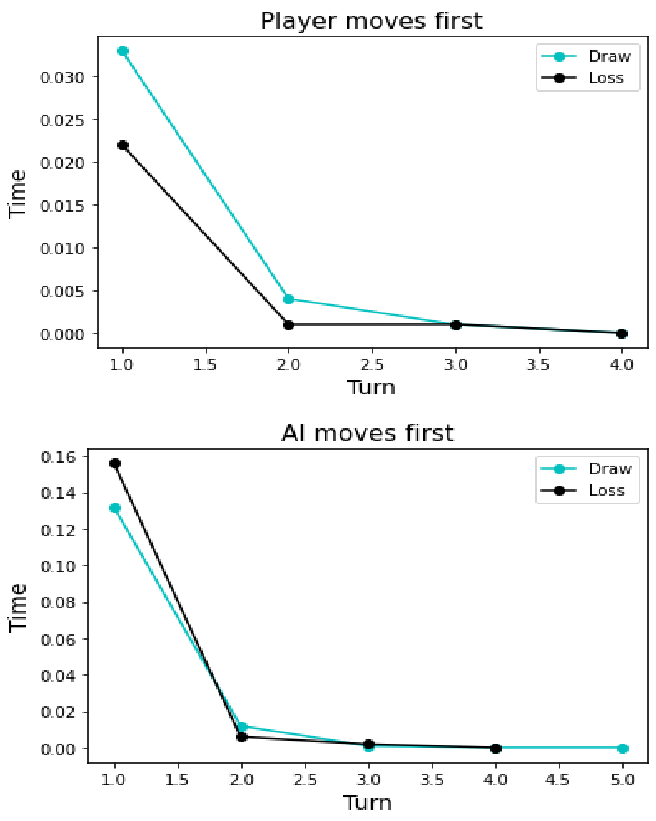


Figure 5. Execution time for alpha-beta AI in tic-tac-toe

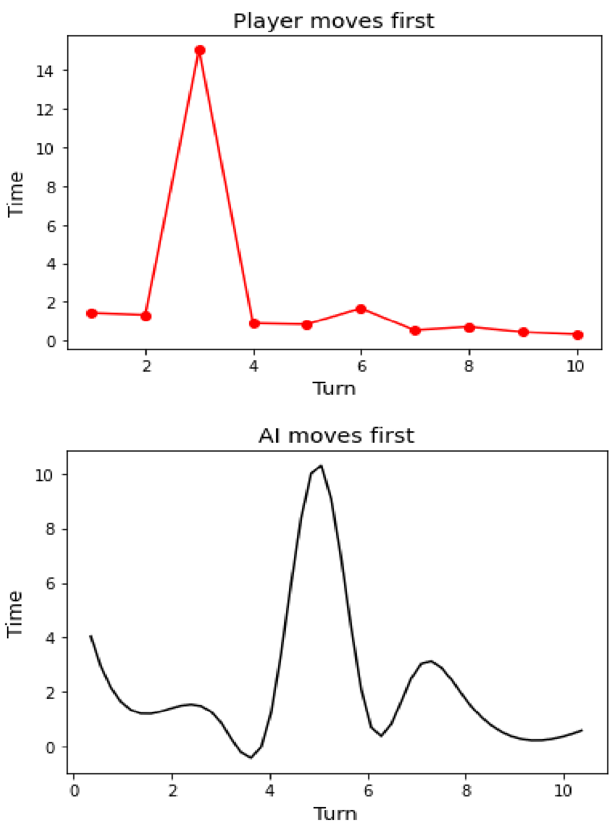


Figure 6. Execution time for alpha-beta AI in connect-4

speeds of the AI in determining which move to make. This measures the length of time it takes for the AI to search the game tree and therefore should get shorter after each move is made.

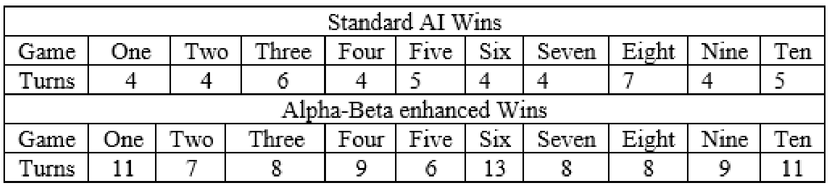
D. Changes to Alpha-beta algorithm

When running tic-tac-toe, I noticed that when the AI began the game it always chose the same position to move, top-left corner. This turn although is the same in every game, it is calculated by the algorithm to be the best scoring move to make. However, this then causes the game to become a bit repetitive and stale as the player will always know where the AI begins. This could also be an advantage for the player perhaps in another board game but not for tic-tac-toe. So why this occurs is due to the search process of the algorithm. For example, the game tree for the first turn will have nine terminal nodes which are the nine empty positions on the board. Since they are all empty and none of the positions provide any advantage over the other, the algorithm chooses the first score it searches i.e., the first position which is [0,0]. To counter this, I used a similar approach to the standard AI model of the game. Using the random function to choose a random position on the board if and only if the board is empty. This was successful as the condition breaks as soon as the first move is made and therefore no more random moves are made for the rest of the game.

V. Results

Here will provide results comparing difficulties against standard and alpha-beta enhanced AI, the rate of wins/draws/losses in both instances from 5 different players, where each player faced both versions of the game AI. As well as the effects of different execution times of each turn of the algorithm for both games.

In this experiment there were 5 players at different skill levels, such as a current university student aged 19, middle aged 48, and a 26-year-old. When observing the results above in figure 4 against the standard AI, although there are noticeable differences between the win rate of the players, it is clear that the players were quite successful in defeating the AI without the assistance of alpha-beta pruning. Studying figure 4 it is easy to see that for the standard AI the win percentage for humans was 80% and the loss was 4%. Although, the loss percentage was almost minimal, it could have possibly been 0%. Perhaps due to loss of concentration, players 2 and 4 lost one game each. On the other hand, when the 5 players went up against the alpha-beta enhanced AI, there was a 0%-win rate and 28% loss rate, where the majority of matches ended up as a draw. This significant difference establishes the fact that the alpha-beta algorithm dominates in games with small number of tree searches. Furthermore, the line graphs above present the time in seconds it takes for the alpha-beta algorithm to find the optimal decision when the AI begins first or when the player moves first. The cyan coloured lines represent a draw whereas the black lines represent loss. When studying both graphs, it is evident that the time it takes for the algorithm to come to a decision, decreases over time, i.e., the first move the AI makes took an average of 0.15s and the final move was made almost instantly at 0s. You will also notice that after the first move is made by the AI, the time the alpha-beta algorithm requires to make the next move significantly declines. For example, in figure 5 when the player moves first, the AI takes an average of



Standard AI Wins										
Game	One	Two	Three	Four	Five	Six	Seven	Eight	Nine	Ten
Turns	4	4	6	4	5	4	4	7	4	5

Alpha-Beta enhanced Wins										
Game	One	Two	Three	Four	Five	Six	Seven	Eight	Nine	Ten
Turns	11	7	8	9	6	13	8	8	9	11

Figure 7. Number of turns needed to win

0.25s to make the first move then the following 3-4 turns takes almost no time at all. This is a result of the tree searching process becoming faster and faster. What this means is that during the first turn for the AI, the algorithm must search every node until the terminal node (end of the tree) for its optimal decision. However, during its next turn there are now less positions on the board and hence less nodes to be searched and evaluated. As a result, the evaluation time for the algorithm becomes shorter and the AI responds faster.

Some results obtained from playing connect-4 are given in the table above, figure 7. The table shows the number of turns required to defeat the AI with and without alpha-beta pruning. Ten games were played each against the standard AI and the alpha-beta enhanced with the same player. Unfortunately, the algorithm was not implemented completely enough to prevent players from winning. However, the performance of the algorithm does show increased difficulty since, on average double the number of turns is needed to defeat the alpha-beta enhanced AI. Also, looking at figure 6 you can see when exactly the algorithm is calculating its next strategic move. For example, at turns 3 and 4 the AI blocks the player from achieving four in a row. The algorithm also takes significantly longer than tic-tac-toe.

When experimenting with different levels of depths to search, I found that the optimal and best performing depth for tic-tac-toe was given as the maximum number of available spaces on the board after every turn during the game. This makes the AI unbeatable and successfully stops a player from winning every game. However, the difficulty of the AI can be reduced to give the player an opportunity to win. This is done by reducing the depth of search to less than 8.

A. *Some drawbacks*

1) *Tic-tac-toe*

Whilst playing several practice matches, I noticed that the algorithm prioritises blocking the human from winning rather than the AI aiming to win itself. When observing figure 5, it is easy to see that the evaluation time taken for the AI's third turn takes slightly longer than its previous and following turns. During this state of the game, the AI had a decision to either win the game or block the player. Winning the game should have had the greater evaluation score during the alpha-beta process, however, did not and hence this is a slight flaw in the program. There are also situations where the AI can win the bout however decides to make a move which can either be advantageous or disadvantageous, elongating the game. For example, in figure 8, the AI can simply move to position [0,2] and win however decides to move to [1,1] Although, the algorithm has these flaws it is still unbeatable and hence the algorithm is undeniably effective.

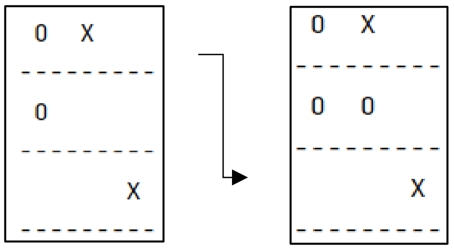


Figure 8. Example moves by AI

The previously mentioned problem of the algorithm choosing the same starting position every time the player moves second, has a complex but easily identifiable solution. "Alpha-Beta does a rigid depth-first, left-to-right traversal of the tree." (Schaeffer and Plaat, 1996). Meaning that the alpha-beta pruning technique will always choose the first node it comes across if there are multiple similar nodes. On the other hand, the SSS* algorithm uses a best-first approach and therefore will always choose a random node if there are identical nodes. Unfortunately, SSS* is very difficult to implement and would not be worth the effort and resources in trying to implement it for a game as simple as tic-tac-toe.

2) *Connect-4*

A problem occurring in the alpha-beta enhanced AI is that the AI is able to detect and stop a player from winning vertically. However, is unable to detect an upward or downward sloping win. The AI can also stop the player from matching four of their pieces horizontally on the first row, but unfortunately has difficulty stopping a horizontal win on any other row. Another drawback is that more than a depth of 4, the alpha-beta algorithm takes a significant amount of time to determine the best move. For example, observe figure 6. There is a sharp increase in the time the AI takes calculating an optimal move for its third and fifth turn. These turns were when the player was about to reach four in a row but is stopped in the AI's next turn. Showing that the algorithm takes some time to determine what the highest scoring move is. Compared to tic-tac-toe, I believe the algorithm takes longer for every turn, due to the greater number of nodes to search.

VI. Conclusions and Future work

It has been 30 years since the alpha-beta algorithm was first discovered and yet there are still many variations just being found. This paper aimed to experiment with different features of the alpha-beta algorithm and determine its best functionality in a game of tic-tac-toe as well as connect-4. Overall, from this experiment it is clear that an alpha-beta enhanced (intelligent) AI performs significantly more strategically than using a random integer generated (standard) opponent. Although the standard AI evaluates its moves faster than the alpha-beta enhanced, i.e., almost instantly, the games are over in a matter of seconds due to the simplicity and randomness of the AI's moves. To further improve alpha-beta's performance a modification I have explored is to allow the algorithm to choose at random a choice from several moves that have the same score value. Rather than the algorithms traditional left-to-right traversal of the tree, where the same move will always be chosen for every game. This addition to the algorithm provides randomness at the beginning of the game, hence challenging players to use different strategies to defeat the AI. This modification gave the AI eight extra positions to begin with, forcing the player to stay focussed if playing multiple consecutive games.

Another feature that could be developed would be to incorporate a heuristic function. Instead of having the algorithm just responding the humans move. The AI could also place moves to encourage confusion in players and make games more difficult. The heuristic could be developed from experience of playing hundreds to thousands of matches. The AI would then teach itself which moves cause humans to play on the attack or defence.

The ability to store the scores of each node in a memory was a feature I attempted to incorporate. Unfortunately, due to time-constraints and complexity I was unable, and hence this could be a future modification to delve into. What this does is enable the algorithm to already know what the best moves are. Therefore, the evaluation times for the AI will be a lot smaller or even instant. This can then lead to the implementation of the MTD(f) algorithm, whose primary function is to store memory. However, as MTD(f) relies on memory it may not perform as well on simple games such as tic-tac-toe where the depth of search is relatively low.

VII. References

Chen J., Lu S., Vekhter D., Strategies of Play, Available at: Strategies of Play (stanford.edu), Website

Java T Point (2011), Available at: Artificial Intelligence | Minimax Algorithm - Javatpoint, Website, **Figure 1,2**

Rashmi J., Hacker earth (March 2017), Available at: Minimax Algorithm with Alpha-beta pruning - HackerEarth Blog, Website

Aske Platt, Jonathan Schaeffer, Wim Pijls, Arie de Bruin, (1994), *'A new paradigm for minimax search'*, University of Alberta, Report, arXiv:1404.1515

Aske Platt, Jonathan Schaeffer, Wim Pijls, Arie de Bruin, *'SSS* = α-β + TT'*, Technical Report EUR-CS-95-02, Erasmus University Repository

Burguillo C. J., (January 2018), *'Game Theory',* Self-organizing Coalitions for Managing Complexity, pp.104-110

Buro M., (January 2002), *'Improving heuristic mini-max search by supervised learning'*, Artificial Intelligence, **Vol. 134**, Issue 1-2, pp. 85-99

Campbell S. M., Marsland T. A., (July 1982), *'A comparison of minimax tree search algorithms'*, Artificial Intelligence, **Vol. 20**, Issue 4, pp. 347-367

Eka Putri S., Tulus, Napitupulu N., (August 2011), *'Implementation and analysis of depth-first search (DFS) algorithm for finding the longest path',* International seminar on operational research (InteriOR), Conference paper

Igor R, Judea P., (October 1982), *'A minimax algorithm better than alpha-beta? Yes and no,'* Artificial Intelligence, **Vol. 21**, Issue 1-2, pp. 199-219

Jonathan Schaeffer, Aske Plaat, (1996), *'New advances in alpha-beta searching'*, Proceedings of the 1996 ACM 24th Annual Conference on AI Science

Kang X., Wang Y., Hu Y., (May 2019), *'Research on Different Heuristics for Minimax Algorithm Insight from Connect-4 Game'*, Journal of Intelligent Learning Systems and Applications, **Vol. 11**, pp. 15-31

Lukas T., Mardi H., Nazori A., (2017), *'The analysis of alpha-beta pruning and MTD(f) algorithm to determine the best algorithm to be implemented at Connect-four prototype'*, Article

Russell S. and Norvig P., (2010) *'Artificial Intelligence: A Modern Approach'*, 3rd ed. Pearson Education, Chapter 5.2 Optimal decisions in games, Book, **Figure 3**

Stockman C. G., (November 1978), *'A minimax algorithm better than alpha-beta?'*, Artificial Intelligence, **Vol. 12**, Issue 2, pp. 179-196

Swaminathan B., Vaishali R., Subashri T. S. R., (November 2020), *'Analysis of minimax algorithm using tic-tac-toe'*, Intelligent systems and AI technology, Article