The Alpha-Beta Pruning Algorithm in Chess

Michael kolawole-Idowu
School of Electronic Engineering and Computer Science - Department of Computer Science
Queen Mary, University of London
London, United Kingtom
m.kolawole@se19.qmul.ac.uk

***Abstract*—A lot of processing time is spent evaluating good possible moves within a game tree, when developing an AI board game. There is a popular technique, the alpha-beta algorithm, which allows the computers to work out the optimal next move, in an efficient fashion, in games such as chess, tic-tac-toe, etc.**

This report would highlight my understanding of the alpha-beta algorithm and would demonstrate how I implemented and integrated the algorithm within a computerised chess game. I then will investigate variations of the application, such as the implementation of a sorting method, to observe whether I could achieve a performance gain. Finally, I will present my findings and conclude on the best approach.

Acknowledgment

I would like to express my deepest great appreciation towards my supervisor, Dr. Matthew Huntbach, for his guidance and support during my final project. I relied a lot on his guidance, and with our unexpected circumstance due to covid-19, communication was difficult, however he still remained a support system to rely on. A massive thank you also goes to my family, friends and God, words cannot express my gratitude enough but thank you.

I. Introduction

1.1. Context & Motivation

I have been fascinated by AI powered computer games and the ways they operate since I was an infant, particularly the concept of playing against "the computer". Artificial intelligence incorporated within the computer gaming industry has consistently shown signs of growth, in popularity and performance. Today, the industry makes up a $152.1 billion global powerhouse (Video Games Industry Statistics, 2020), and algorithms are constantly evolving to better mimic human behavior.

In 1997, an IBM AI chess program named Deep Blue had played reigning chess world champion, Kasparov, and won, becoming the first computer to beat a world champion chess player (IBM, 1997). The Deep Blue chess program incorporated the alpha-beta search algorithm. The algorithm performs particularly well with games like Chess, tic-tact-toe and connect4, whereby they are with a fixed board and pieces, fixed set of rules, no random elements, each player taking turns to move a piece on the board, and each player can see all elements of the board and pieces.

As I have been keen to learn the game of chess at a more intermediate level, I wanted a game that was difficult to beat, using an algorithm that finds out the optimal next move in a game. I decided to build an AI Chess program that allows a human to play against "the computer", implemented using the alpha-beta search algorithm.

1.2 Aim

An essential aspect of this project would be the understanding of this algorithm and to implement it for my chess program. This report would detail my understanding and development of the algorithm and the chess application. The three main sections that would demonstrate my progress throughout this project, are found within the background research, methodology and results sections. These three sections display a progressive account of my project.

The overall goal is design and develop a working Chess interface where humans play against the computer, implemented with the Alpha-beta algorithm.

Objectives:

- Research the Alpha-beta algorithm but also other algorithms that are used for turn based, board games (particularly chess) and how they can be incorporated.
- Research approaches of displaying GUI; such as the board, piece, messages for users etc.
- Research techniques that can be applied within the algorithm to make it more efficient, speeding up the AI.
- Apply the knowledge gained by implementing and developing an optimised AI chess application
- Test and compare results

II. BACKGROUND RESEARCH.

The focus of this chapter is to provide an overview of background research and academic literature used to support this project. The central focus of this paper, and this chapter, would be examining research surrounding the Alpha-Beta search algorithm. This section would also review prior discussions on surrounding topics, such as the Minimax algorithm, Artificial intelligence more broadly and the implementation of the algorithm, specifically with a complex game like chess.

2.1 History

In 1944, a publication by von Neumann and Morgenstern introduced and described the idea and process called minimax, and how this algorithm could be used to identify the final and best outcome for two-person zero-sum games, like chess (Kjeldsen, 2001). By 1956, an advancement of the minimax algorithm had been developed and proposed by John McCarthy during the Dartmouth Conference, and he called it the Alpha-Beta algorithm. The algorithm is enhanced as it looks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. The algorithm stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move.

2.1.1 John McCarthy & Artificial intelligence

John McCarthy, was an American computer scientist and a pioneer in the field of artificial intelligence. In 1955, McCarthy coined the term Artificial intelligence, and made many developments within this field, as well as the introduction of the Alpha-beta algorithm.

In the late 1950s, computer enthusiasts were optimistic and had thought true Artificial intelligence was on the brink of being accomplished, whereby computers could think like humans. However, McCarthy giving a presentation in 2006 entitled: "HUMAN-LEVEL AI IS HARDER THAN IT SEEMED IN 1955", identified a disappointing progression within the field. McCarthy stated, “I hoped the 1956 Dartmouth summer workshop would make major progress. … If my 1955 hopes had been realized, human level AI would have been achieved before many of you were born…. My 1958 ‘Programs with common sense’ made projections that no-one has yet fulfilled.” (McCarthy, 2006).

However, McCarthy also discusses how chess programs accomplish some level of AI and human play, particularly implemented with Alpha-beta pruning algorithm, stating how “Chess programs catch some of the human chess playing abilities but rely on the limited effective branching of the chess move tree…. Alpha-beta pruning characterizes human play, but it wasn't noticed by early chess programmers” (McCarthy, 2006).

2.2 CHESS

Chess is a two-player board game, played with a chessboard and sixteen pieces of six types for each player, one player plays as white and the other as black, as seen in Figure 1. The aim of the game is for a player to threaten with inescapable capture of the opponent’s king (checkmate).

However, chess games often do not end in checkmate, as players often quit if they feel they are on the verge of losing, and a game can also end in a draw or stalemate in many ways too.

Piece	King	Queen	Rook	Bishop	Knight	Pawn
Number	1	1	2	2	2	8
Symbols	♔ ♚	♕ ♛	♖ ♜	♗ ♝	♘ ♞	♙ ♟

Figure 1 image of pieces (Rules of chess, 2020)

2.2.1 Game play

The player controlling the white pieces commences the game and moves first, then each player will alternate moves. A player is required to make a move on their turn at all times as it is not legal to skip a move, even with moves that may be detrimental. A fundamental strategic rule is to capture the opponent's pieces while attempting to preserve your own. Play continues until a game is resolved in the ways mentioned earlier.

starting position chess board (Chess Pieces: Board Setup, Movement, and Notation, 2020)

Basic move of play available to each piece (Schiller, 2003.):

- The king moves exactly one square horizontally, vertically, or diagonally. However, a special move with the king known as castling is allowed only once per player, per game.
- A rook moves any number of vacant squares horizontally or vertically. (It also is moved when castling.)
- A bishop moves any number of vacant squares diagonally.
- The queen moves any number of vacant squares horizontally, vertically, or diagonally.
- A knight moves to the nearest square not on the same rank, file, or diagonal, i.e. in an "L" pattern. The knight is not blocked by other pieces: it jumps to the new location.
- Pawns have the most complex rules of movement:
    - A pawn moves straight forward one square, if that square is vacant. If it has not yet moved, a pawn also has the option of moving two squares straight forward,

provided both squares are vacant. Pawns cannot move backwards.
    - Pawns are the only pieces that capture differently from how they move. A pawn can capture an enemy piece on either of the two squares diagonally in front of the pawn (but cannot move to those squares if they are vacant).

2.2.2 Additional considerations -

- The official chess rules do not include a procedure for determining who plays as which piece and decisions are often left with the opponent.

This consideration meant for designing a program for this project which allows for both "the Computer" and the Human player the opportunity to adopt either a black or white piece, with the human deciding prior to the game starting.

- Games are also often played under a time controlled setting, where a player who exceeds their time limit loses the game.

Implementing a program that strictly achieves a time controlled setting may be deemed as impractical and unenjoyable for a computer game designed for leisure. However, it supports the need for a program implemented with an efficient and well performing algorithm to power it, such as the alpha-beta algorithm.

2.2.3 Basic Strategy

Generally speaking, the more pieces a player has or an aggregation of more powerful pieces, often mean a greater chance of victory. As each piece type has its own restrictions, chess pieces are given their own relative value. "Chess piece values indicate the value of the different chess pieces and how they relate to each other" (Chess Pieces Value - Chess Terms, 2020). Calculating this value of each piece provides some idea of the appropriate decision to be made when making a move and capturing the opponent's pieces. Illustrating a need for a sufficient evaluation function within a chess engine is crucial for the overall gameplay.

2.3 The Standard valuations

The "Standard valuations" is the most common value assignment used within chess (Capablanca and Firmian, 2006) and was also adopted and evolved within this project.

The "Standard valuations" values each piece as such (Capablanca and Firmian, 2006):

- The King - is undefined as it attempts to remain uncaptured, although Chess engines usually assign the king an arbitrary large value to indicate that the inevitable loss of the king (due to checkmate) trumps all other considerations (Levy and Newborn, 1991).
- The queen - 9 points
- The rook - 5 points
- The bishop - 3 points
- The knight - 3 points
- The pawn - 1 point

This valuation system has worked well within chess and has been implemented for centuries, however it has also shown signs of shortcomings and a need for flexibility. For instance, depending on the game situation and the piece positions, the exact piece values can differ considerably throughout the game (Chess Pieces Value - Chess Terms, 2020). An example of this can be seen in the endgame situation, with less risk of checkmate, the fighting value of the king is actually considered around four points (Lasker, 1988).

2.4. Main supporting literature

Russell and Norvig's literature, Artificial intelligence a modern approach, was used to support many decisions in implementing an Alpha-beta algorithm for an AI chess game. Russell and Norvig identified an efficient need for the alpha-beta pruning algorithm, particularly in a game such as chess whereby "they usually have time limits, penalizing inefficiency very severely" (Russell and Norvig, 2010), as mentioned above. They begin to introduce the concept of pruning and how it "allows us to ignore portions of the search tree that make no difference to the final choice, and heuristic evaluation functions allow us to approximate the true utility of a state without doing a complete search" (Russell and Norvig, 2010). This displayed a necessary need and benefit from adopting and implementing a pruning algorithm.

2.4.1 Minimax

An understanding of the minimax algorithm is vital to understanding the implementation of the alpha-beta pruning algorithm. Russell and Norvig discuss how the algorithm is a depth-first search algorithm, which creates a performance inefficiency impractical, but also serves as a manageable method for game play.

Describing a 5-step process used by the minimax algorithm "to determine the optimal strategy for MAX, and thus to decide what the best first move is", which is to (Russell and Norvig, 2010):

- Generate the whole game tree, all the way down to the terminal states.
- Apply the utility function to each terminal state to get its value.
- Use the utility of the terminal states to determine the utility of the nodes one level higher up in the search tree.
- Continue backing up the values from the leaf nodes toward the root, one layer at a time.
- Eventually, the backed-up values reach the top of the tree; at that point, MAX chooses the move that leads to the highest value.

Russell & Norvig state how the minimax algorithm makes a non-practical assumption that the program has time to

search all the way to terminal states and how a heuristic evaluation function should be applied to enhance the algorithm, determining how good a game state is for a given player.

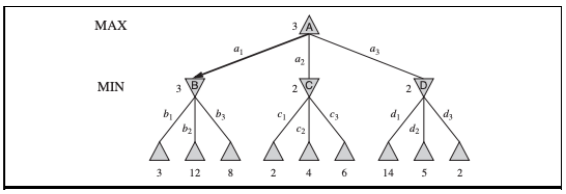

Figure 3: Simple minimax game tree (Russell and Norvig, 2010 pp 164)

2.4.2 Evaluation Function

Numerical values are assigned to game states, for example chess pieces within a chess game. This leads to pieces being able to be compared against each other within a game. It is stressed the importance of the quality of a program's evaluation function, and how game-playing performance is extremely dependent on it. "If it is inaccurate, then it will guide the program toward positions that are apparently "good," but in fact disastrous" (Russell and Norvig, 2010).

Because of the shortcoming observed above from the Standard valuations, the "Simplified Evaluation Function", published by Tomasz Michniewski, was adopted within this project instead. The "Simplified Evaluation Function" (Michniewski, 1995) takes the "Standard valuations" but also adopts some additional features, such as giving bonuses for pieces positioned well and penalties for pieces positioned badly (Michniewski, 1995).

The evaluation function also sets out to:

- Avoid exchanging one minor piece for three pawns.
- Encourage the engine to retain both the bishops.
- Avoid exchanging two minor pieces for a rook and a pawn.
- Stick to human chess experience. (Simplified Evaluation Function - Chessprogramming wiki, 2020)

2.4.3 Alpha-beta Pruning

Illustrated within figure 4, the effectiveness of applying the Alpha-beta Pruning algorithm to a standard minimax tree is demonstrated, as it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision (Russell & Norvig, 2010). This pruning algorithm can be layered with the minimax algorithm to achieve a more efficient search, and Russell & Norvig, describes the "search function itself as just a copy of the MAX-VALUE function with extra code to remember and return the best move found" (Russell & Norvig, 2010).

```
function MAX-VALUE(stategame, a, β) returns the minimax value of state
  inputs: state, current state in game
          game, game description
          a, the best score for MAX along the path to state
          β, the best score for MIN along the path to state

  if CUTOFF-TEST(state) then return EVAL(state)
  for each s in SUCCESSORS(state) do
     a ← MAX(α, MIN-VALUE(s, game, a, β))
     if a ≥ ft then return ft
  end
  return a

function MIN-VALUE(state, game, a, ft) returns the minimax value of state

  if CUTOFF-TEST(state) then return EVAL(state)
  for each s in SUCCESSORS(state) do
     ft ← MIN(ft, MAX-VALUE(s, game, a, β))
     if ft < a then return a
  end
  return ft
```

Figure 4 - The alpha-beta search algorithm, it does the same computation as the minimax, but prunes the search tree. (Russell and Norvig, 2010 pp 132)

2.5 SSS* search algorithm

Other algorithms and variations on the basic algorithm were also explored and considered within this project; particularly the SSS* search algorithm. The SSS* search algorithm was introduced in 1979 by George Stockman, and is based on the notion of solution trees, an algorithm established on state space search for computing the minimax value of game trees (Stockman, 1979).

"It builds a tree in a so-called best-first fashion by visiting the most promising nodes first. Alpha-Beta, in contrast, uses a depth-first, left-to-right traversal of the tree" (Plaat, 1995). Stockman and others consider the algorithm to be superior to the Alpha-beta algorithm, as the SSS* is deemed more efficient, in the sense that the SSS* algorithm never evaluates a node that Alpha-beta can ignore, and may prune some branches that the alpha-beta algorithm would not.

Igor Roizen and Judea Pearl (1983), however, argued "the disadvantages of SSS* have proven a significant deterrent in practice" and characterized the general view of SSS* as:

1. it is a complex algorithm that is difficult to understand,
2. it has large memory requirements that make the algorithm impractical for real applications,
3. it is "slow" because of the overhead of maintaining the sorted OPEN list,
4. it has been proven to dominate Alpha-Beta in terms of the number of leaf nodes evaluated, and
5. it evaluates significantly fewer leaf nodes than Alpha-Beta.

This demonstrates that the number of positions saved from evaluation by the SSS* compared to the alpha-beta algorithm is limited and generally not worth the increase in resources.

III. DEVELOPMENT

This section will describe the design and development process of my chess application. I will be discussing how I approached the creation of the AI game; laying out my early design decisions, as well as the GUI development. I will

then focus on the implementation of the alphabeta chess game, along with the integration of my sorting method used for performance optimisation.

The development process would be split into three subparts:

1. The initial design process and the chess game GUI.
2. Basic functioning and the implementation of the alpha-beta algorithm.
3. Sorting optimisation technique.

3.1 Design and GUI

As the central focus of this project laid mainly with the algorithm powering the chess game, our chess game features a simple graphical user interface. The GUI was built using java Swing, a lightweight GUI toolkit written entirely in java, which has a wide variety of widgets for building optimized window based applications (Waseem, 2020). This was used to create the containers for the visual display of the board.

The implementation process for the application began with developing a 500px by 500px Jframe, implemented using a for-loop statement to create the checkered board for the visual display of our game board. In order to also develop my chess game, an image of the chess pieces was also required (figure appendix). It was necessary to select an image, where each piece was all equally sized and spaced the same (32x32px for each piece). This was to allow for the correct selection and display of the pieces. After each image of a piece is identified by the application, I then integrated the chess piece image with my checkered board, using a for-loop with a switch statement inside to place each piece in its correct spot on the board. Combining these elements was used to visually develop our chess game, creating the board along with the pieces, as seen within figure 5.

Figure 5 – Final visual display of my chessboard, featuring the checkered board, along with pieces on the board.

Having developed the GUI, it was then required to create the basic functioning features to allow restricted moves for each piece.

3.2 Basic functioning

My application uses three fundamental classes that all interact with one another to represent the chess game.

1.The Board class – (not to be confused with visual board GUI) represents the computerised board as a single two-dimensional 8x8 array, storing information of the location of each piece throughout the game. Capital characters representing white pieces and lowercase characters representing black. In place of having just an array representing the board position, the Board class uses an object with an array inside it. It makes a new object with a copy of the array and moves the objects in the copy, instead of moving the actual elements in the array. This was later adopted within the updated version of the game as it leads to cleaner more general code, however, this method I predict could have a negative effect on the efficiency due to performing the copying (as seen in the results section).

The board class also contains a kingPosC() method, as well as information such as the depth used for the Alphabeta algorithm. The kingPosC() method is vital for move generation to prevent a move that would put the king in danger as it keeps a tab on the location of both Kings.

2. The pieces' class - represents each piece in the game and each possible move that is accessible to them. Each piece uses their own for-loop with an if-statement inside it to map out their possible moves, it also works out the piece's surrounding squares and figures out if the chosen spot is available and whether the piece is able to move there. Each piece also uses an "oldPiece" variable which stores information about whether there is an opponent piece in a spot wanting to be moved to. If the opponent piece is able to be captured, it removes the old piece and replaces it with the piece that has captured it.

Each piece also calls on a kingSafety() method, which interacts with the kingPosC() method in the board class to ensure a move is valid before making it and again ensuring the move doesn't place the king in danger.

3. The Move class provides the actual mechanics to perform the movement for each piece, after evaluating the factors from the classes above. The class uses 2 methods, the makeMove and undoMove methods, which is essential to my engine but possibly may not be the most efficient approach. Although trying to mimic a human gameplay, unlike a human, when evaluating decisions my engine cannot "imagine" moves, so instead the engine has to make the moves on the board and then undo those moves to resume back to its original position so that it can think of multiple lines of play.

An alternative approach considered to overcome this weakness, and making the undoMove redundant, was to not make changes to the actual board but rather create a new board representing the change. However due to limited time and resources I was unable to further explore and implement this approach, although it would be considered for further improvements of the application.

3.3 Rating

Building my evaluation function, it was important that I maintained a good balance between having a sufficient

rating system and having a system too powerful, which could negatively affect the performance of the game. There were several key rating categories (methods) to factor in, as it was important for our evaluation function to consider multiple perspectives for a more advanced chess game

I decided to spit the rating into 4 methods:

The **RateMaterial()** method rates each piece using a switch statement. The method adopts and enhances the values used by the standard valuation (Capablanca and Firmian, 2006) to score each piece, with each piece giving a score of:

- The queen - 900
- The rook - 500
- The bishop - 300
- The knight - 300
- The pawn - 100

However, as mentioned in my background research section, I discovered a major drawback surrounding the bishop material that should be accounted for. That is, the bishop piece is relegated to only one square-color and is therefore limited to half the squares on the board (Sopher, 2020). Consequently, losing one of your bishops would and should have a larger negative impact than the loss of one of your knights, as half of the board becomes unavailable from the bishop's perceptive. Our alpha-beta chess game accounts for this using an if-statement to update the rating depending on the number of bishops in game.

RateMovability() - This method is used to assess whether the game state is in check, checkmate, stalemate, and how flexible it is to make a move. The method allows our program to attempt to force the opponent's move and for our computer to attempt to avoid losing, giving our engine a human-like gameplay as it communicates that the goal of the game is to checkmate, rather than just capturing pieces.

RatePositional() - Mentioned earlier within the background research section of this paper, The "Simplified Evaluation Function" by Tomasz Michniewski (Michniewski, 1995) was adopted to rate the position of each piece on the board. The method interacts with different two-dimensional 8x8 arrays, which mimic the board, scoring each piece depending on where they are on the board. The method also interacts with 8x8 arrays that considers the King's position in the mid/end game and acknowledging the king becomes more powerful in its attack in the latter stages, giving an increased rating to the king.

RateAttack() - This method evaluates pressure based on each piece's value, allowing for a more intuitive game-play. This allows the computer to assess where danger is coming from, recognizing that playing a piece in a square could lead to its capture.

3.4 Alpha-beta Algorithm

The AlphaBetaAlgorithm class along with the rating class, our evaluation function, is responsible for our chess game AI. Alpha-beta pruning as mentioned earlier is an approach of finding the optimal solution while avoiding searching branches of moves which won't be selected. There are two kinds of nodes in a search tree for a two-player game, nodes representing one player's moves and nodes representing your opponent's moves.

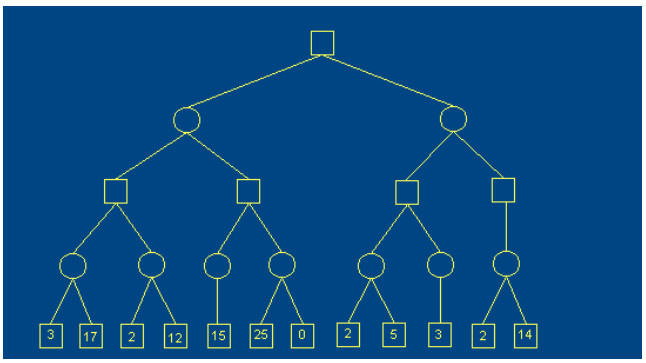

Figure 6 - Nodes representing your moves are generally drawn as squares and circles as your opponents (CS 161 Recitation Notes - Minimax with Alpha Beta Pruning, 2020).

Applying the figure 6 to a chess game, each node represents a chess position, the goal at a MAX node is to maximize the value of the subtree rooted at that node and the goal at a MIN node is to minimize the value. The figure like our game has a depth of 4, attempting to find the optimal move by looking 4 moves ahead, two moves for myself and 2 for my opponent. Once our first node reaches a depth 4, we then run our evaluation function, rating each move by its outcome (In figure 6, the first node has a value of 3). Alpha-beta pruning works by keeping track of the alpha, beta and evaluation values, pruning branches not containing the optimal, and theoretically trimming moves within it that are bad moves and unlikely to occur. The algorithm I implemented was developed somewhat specifically with our chess game in mind, returning an alpha, beta, evaluations value, as well as also returning the move.

With that in mind my AlphaBetaAlgorithm rather than returning an Integer for instance, it returns a String value. The algorithm required multiple variables to make it function, starting with an int depth variable, along with an int beta, int alpha, String move and int player variable was also required.

- **Depth** - to represent how far ahead our program should search to find the optimal move
- **Beta** - to represent the minimum upper bound of possible solutions (CS 161 Recitation Notes - Minimax with Alpha Beta Pruning, 2020).
- **Alpha** - to represent the maximum lower bound of possible solutions (CS 161 Recitation Notes - Minimax with Alpha Beta Pruning, 2020).
- **Move** - to return an optimal move
- **Player** - to represent which player, 1 representing ourselves and 0 representing the computer.

3.4.1 How Alphabeta produce Moves

The algorithm begins by interacting with the pieces' class by producing a String variable, "list", which produces a list of all possible moves. We then must ask a question by using an if-statement, of what we must do once when we reach the maximum depth or if the list.length() method is equal to 0 (no possible moves available), which is for the algorithm to stop searching and to deliver the optimal move. The optimal move is returned, as well as a rating and player to negate the score every second depth. The algorithm then goes on to incorporate out sort method for performance optimisation, however this would be further explained in the sorting optimisation section below.

The program uses a For statement to loop through, grabbing a "batch" of 5 possible moves, making the moves and then rotating the board to make a move for the opponent, using our flipBoard() method.

```
// basic alpha-beta search
int alphabeta(int depth, int alpha, int beta)
{
    move bestmove;
    if (game over or depth <= 0) return winning score or eval();
    for (each possible move m) {
        make move m;
        score = -alphabeta(depth - 1, -beta, -alpha)
        if (score >= alpha) { alpha = score; bestmove = m; }
        unmake move m;
        if (alpha >= beta) break;
    }
    return alpha;
}
```

Figure 7 – Alpha beta search pseudocode (ICS 180, February 2, 1999, 2020)

The pseudocode in figure 7 was adopted to assist in producing my code for the algorithm, and similar to the pseudocode, my code also produces a String that calls on itself. This technique of recursion, calls on the AlphaBetaAlgorithm class, however with a depth of -1, meaning it could run only so often before reaching a depth of 0 and have to produce the optimal move.

The code then produces the move as well as the value, and rotates the board back and unmakes a move after.

3.4.2 How Alphabeta produce Values

The explanation above expands on the approach implemented to return the optimal move, however the next section would explain how our Algorithm produces the values, which are also required.

To produce values, which adapts the maximiser and minimiser nodes for alpha-beta pruning, our program has to first workout whose turn it is as this would have an effect on the results. After the child nodes have been examined, the return value from the child nodes is then compared with either the alpha in the maximiser nodes or beta in the minimiser nodes. If the new values are preferred, they are then updated, as well with either the alpha or beta value with the value just returned.

The new alpha or beta value is then later used when examining further child nodes, and an inspection for pruning then occurs. The pruning is executed when the alpha value is greater than or equal to the beta value, and the optimal move found is then returned.

3.5 sorting optimisation

This approach to providing optimisation to my AI chess game was a simple one in the improvement of the game's performance. A sorting method was adopted, as this method delivers a more efficient way of producing the optimal position (as seen in the results section) by attempting to sort from the best possible move to the worst. The result shows how the better each possible move is sorted, the better our alphabeta algorithm performs by analysing faster and pruning quicker.

This method, for the consideration of performance, only picks out up to 6 best possible moves to sort, and then leaves the remaining other moves within the list unsorted. Firstly, as the best possible move is unknown to us at the point of sorting, the method does it's best to make some educated predictions, and if correct, would not require to search the remaining unsorted list of possible moves, thus performing faster.

Taking this into account this method uses an integer array to produce its own rating list based on our evaluation function in the rating class to evaluate whether a move is simply a good move or a bad move. A for-loop is then used to split the possible moves into two lists; newListA, which contains 6 of our "best" moves, and newListB which contains all our next possible moves removing the 6 from newListA. After the two lists have been developed, they are then combined with the best 6 sorted at the top of the finalised list. The method is then integrated with the system and placed within the alphaBetaAlgorithm class to provide optimisation.

IV. Testing and Results

Within this section I will be discussing both the testing phase, which took place to ensure our chess game was operating as expected, and the results, comparing our different updated versions of the game.

The results sub-section, would be comparing our original version of our game with no sorting method integrated against the same version with the sorting integrated, to assess performance optimisation. I will be also comparing our new board function against the original version to again compare the performance, as well as also observing the performance after integrating the sorting function too.

4.1 Move generation testing

It was essential after developing each move generation to test accordingly that the move is properly implemented. Having our pieces interact with the chessboard, our 8x8 array within the board class, along with interacting with the possible moves methods, I was able to assess if a move was implemented correctly.

Our possible moves method prints onto the console how many moves are available, where on the board these moves are available and what pieces are available for capture. To test each move generation, I would place each piece around

the board in specific positions and explore each possible move and what pieces are available to them. Once performing each move around the board and assessing whether it is a valid move, they were then accepted and the game was updated. I also then tested illegal moves to ensure they were not included in our game, such as assessing blocked moves and also ensuring no pieces are able to perform a move if the king was not safe.

4.2 Alpha-beta pruning testing

Ensuring my Alpha-beta pruning algorithm was implemented and working correctly was immensely integral to the success of my project. Without a working algorithm then the previous work completed would become somewhat redundant.

In order to test my algorithm, an external resource was used to support my test. Using illustrations of alpha–beta pruning game tree, figures 8 & 9, I was able to explore whether my algorithm was working correctly or not. The figures illustrate which sections of the tree are expected to be pruned. These diagrams work by acknowledging that the bundle of subtrees collectively returns the value of an equivalent subtree or worse, and therefore cannot influence the final result.

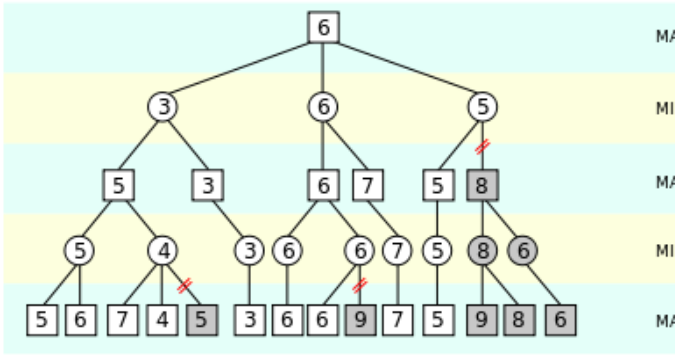

Figure 8 – Alpha beta game tree test 1 (Dailly, Gotojuch and Henning, 2020)

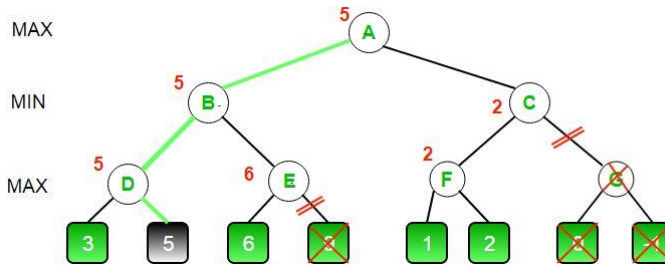

Figure 9 – Alpha beta game tree test 2 (Minimax Algorithm in Game Theory | Set 4 (Alpha-Beta Pruning) - GeeksforGeeks, 2020)

To artificially recreate these subtrees, I was required to manually input each rating value, in an attempt to return the final value at the top of the figures. The scanner method was used within the rating class, in order to take in an integer input that was to represent the score. I then used a scanner method again within the alpha-beta pruning algorithm class, to artificially input the number of possible moves.

Interacting with our algorithm, as well as also matching our depth to the depth found in the figures, I am able to begin my test.

I would then work down the tree, inputting the number of possible moves and the scores, down one node till reaching the evaluation function. After reaching the evaluation function and inputting a score, the test would then ask again for how many possible moves are available and after placing an arbitrarily high value, my program is then expected to ignore this, arbitrarily high value and begin to prune away at the tree.

My program as expected, matched what was found in the figures and instead of searching up to that arbitrary value, it replicates the diagrams and prunes off parts of the tree. After working my way round the tree, and inputting the last value my test again matches the diagrams as expected and stops running, returning the final value at the top of the tree.

This test demonstrated our algorithm is working correctly, not examining whether the game can beat a human but rather testing whether all the components work together.

4.3 Sorting optimisation test and results

After developing and testing my alpha beta chess game to ensure it was working correctly, I then attempted different variants of the program to test whether I could gain a performance edge.

In this section, I would be testing and comparing the results found from the implementation of our new sorting technique, as well as, the effect of implementing a new board object technique, of making a new board object with a copy of the array and moving the objects in the copy.

To perform the test, a timer was integrated within the system to measure how long the computer takes to come up with the first move, after using the algorithm to calculate it. I decided to perform the test after the first move, as the program tends to run faster after the first move due to the number of possible moves decreasing and the pruning of our game tree.

I performed the test five times for each variant, and also included an average processing time and these were the results:

	Original - With No sorting	Original - with sorting	New board and with no sorting	New board and with sorting
T1	6301 milliseconds	2770 milliseconds	5448 milliseconds	2920 milliseconds

T2	8200 milliseconds	2911 milliseconds	5189 milliseconds	2768 milliseconds
T3	6233 milliseconds	2829 milliseconds	5628 milliseconds	1983 milliseconds
T4	6950 milliseconds	2933 milliseconds	6488 milliseconds	2741 milliseconds
T5	6876 milliseconds	3159 milliseconds	6148 milliseconds	2665 milliseconds
Avg time	6912 milliseconds	2920 milliseconds	5780 milliseconds	2615 milliseconds

Figure 10 – Alpha beta search pseudocode (ICS 180, February 2, 1999, 2020)

As seen from the results from figure 10, the sorting optimisation technique implemented provided an enormous performance increase compared to the first variant without the sorting technique implemented. There was a decrease on average of 57.75 % of thinking time with the sorting technique implemented, which was a great sign of success from the technique.

The new board object technique, provided a surprising outcome. Comparing like for like, I expected there to be an increase in processing time, comparing the original with no sorting and the new board technique with also no sorting. This prediction was due to the program creating a copy of the array, however there was instead a decrease in processing time of 16.38%. This decrease was most possibly due to the advantages of this technique, of leading to cleaner and more direct code.

Finally, the figure also shows how the implementation of the sorting method along with our new board technique, provided the biggest decrease in processing time and was our fastest program. With an average of 2615 milliseconds, this final version provided a 56.76% decrease against the new board technique with no sorting technique implemented, and an astonishing 62.17% decrease against our original with no sorting or new board techniques implemented.

These were all promising signs of an improved application from adopting these new techniques. However, as this was a small and limited test, further improvement could also be adopted in taking this project even further. For example, an adaption of new board technique was considered, whereby the program does not change the actual board but instead to create a new board representing the change. As mentioned earlier in the essay, this would therefore make the undoMove method redundant, and hopefully have a positive effect on the program performance.

V. Conclusion

In conclusion, I am very pleased with the progress of my project and the application developed. I learned a lot regarding the implementation of an algorithm within a computerised board game, and the use of a lot of the skills gained within different areas. My understanding of graphical user interface, sorting techniques, algorithms particularly the alpha beta pruning algorithm, and many more areas, have all grown immensely.

I had gained some knowledge of object oriented programming, simple algorithms, abstract data structures and software engineering, prior within my course. However, I began this project with no prior experience or knowledge of the development of AI games. With the assistance of my supervisor, Matthew Huntbach, and my background research I was able to develop my understanding of how the alphabeta trees operated and how it could be implemented to produce an advanced AI game. Intensive research on how algorithms work and could be incorporated with a chess game was an enjoyable experience and led to some interesting conclusions going into the design process.

The implementation phase was a complex and difficult period, and again without the support of my supervisor, I felt somewhat incapable at times of completing the project. However, after tireless efforts, seeing the progression of my chess game was an extremely rewarding feeling. From developing a sufficient GUI, to developing the board representation, to developing the moves generation, to developing the algorithm, was all truly satisfying. As well as the development of my AI chess game, I am also pleased with the results of my optimisation techniques. The final optimised chess game compared the first original version saw an improvement of 62.17% faster on average, which is an incredible outcome.

VI. Future Work

I am pleased with my end result, however there are a few areas in which could be improved or further experimented, with more time and resources.

Firstly, I will continue to look to improve the GUI, by developing a more user-friendly interface and include better features for user experience. I would like to design a more sophisticated GUI, such as including a notification pop up box to announce when the king is in check or mate. Also, announcing when the game is a draw or stalemate.

I also noticed a few bugs that I have not been able to resolve as of yet but I will continue to work on, which is when the computer is chosen to start the game it plays as black, rather than starting as white. As well as that I was also unable to develop code to allow the rook (castle) and king to perform “castling”. These both were due to this being more advanced development, and as I had little experience with java and lack of resources (due to the covid-19 lockdown), this made it difficult to further complete my implementation.

In the future, I would also like to experiment with multiple variants to assess the outcome. Experimenting with other algorithms, such as SSS* algorithm and the Monte Carlo algorithm, as well as enhancing our evaluation function, would be interesting to observe the effect on the performance and game play.

VII. References

- Capablanca, J. and Firmian, N., 2006. Chess Fundamentals. New York: Random House.
- Chess.com. 2020. Chess Pieces Value - Chess Terms. [online] Available at: <https://www.chess.com/terms/chess-piece-value> [Accessed 19 July 2020].
- Chessprogramming.org. 2020. Simplified Evaluation Function - Chessprogramming Wiki. [online] Available at: <https://www.chessprogramming.org/Simplified_Evaluation_Function> [Accessed 19 July 2020].
- Web.cs.ucla.edu. 2020. CS 161 Recitation Notes - Minimax With Alpha Beta Pruning. [online] Available at: <http://web.cs.ucla.edu/~rosen/161/notes/alphabeta.html> [Accessed 30 July 2020]
- Dailly, P., Gotojuch, D. and Henning, N., 2020. *Figure 3.3: An Example Of Alpha-Beta Pruning [1]*. [online] ResearchGate. Available at: <https://www.researchgate.net/figure/An-example-of-alpha-beta-pruning-1_fig3_268426213> [Accessed 7 August 2020].
- Waseem, M., 2020. Swing In Java: Creating GUI Using Java Swing | Edureka. [online] Edureka. Available at: <https://www.edureka.co/blog/java-swing/> [Accessed 27 July 2020].
- Www-formal.stanford.edu. 2006. McCarthy. [online] Available at: <http://www-formal.stanford.edu/jmc/slides/wrong/wrong-sli/wrong-sli.html> [Accessed 19 July 2020].
- GeeksforGeeks. 2020. *Minimax Algorithm In Game Theory | Set 4 (Alpha-Beta Pruning) - Geeksforgeeks*. [online] Available at: <https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/> [Accessed 7 August 2020].
- IBM. (n.d.) Deep Blue. Available from: http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/ [Accessed 20nd July 2020]
- Ics.uci.edu. 2020. ICS 180, February 2, 1999. [online] Available at: <https://www.ics.uci.edu/~eppstein/180a/990202b.html> [Accessed 31 July 2020].
- Igor Roizen and Judea Pearl. A minimax algorithm better than alpha-beta? Yes and no. Artificial Intelligence, 21:199–230, 1983.
- Kjeldsen, T., 2001. John von Neumann's Conception of the Minimax Theorem: A Journey Through Different Mathematical Contexts. Archive for History of Exact Sciences, 56(1)
- Lasker, E., 1988. Lasker's Chess Primer. London: Portman Press.
- Levy, D. and Newborn, M., 1991. How Computers Play Chess. New York: Freeman u.a.
- Norris, J., 2020. UC Berkeley Launches Center For Human-Compatible Artificial Intelligence. [online] Berkeley News. Available at: <https://news.berkeley.edu/2016/08/29/center-for-human-compatible-artificial-intelligence/> [Accessed 19 July 2020].
- Michniewski T., Samouczenie Programów Szachowych. Uniwersytet Warszawski, 1995
- Russell, Stuart J.; Norvig, Peter., 2010. Artificial Intelligence: A Modern Approach (4rd ed.). Upper Saddle River, New Jersey: Pearson Education, Inc.
- Schiller, Eric., 2003. The Official Rules Of Chess. Cardoza Publishing,U.S.; 2nd Revised edition edition
- Sopher, P., 2020. Who Wins The Bishop-Knight Exchange?. [online] The Atlantic. Available at: <https://www.theatlantic.com/entertainment/archive/2014/11/knight-vs-bishop/383202/> [Accessed 2 August 2020].
- Stockman, G., 1979. A minimax algorithm better than alpha-beta?. Artificial Intelligence, 12(2), pp.179-196.
- TechJury. 2020. Video Games Industry Statistics [online] Available at: <https://techjury.net/blog/video-games-industry-statistics/#gref> [Accessed 21 July 2020].
- Thechesswebsite.com. 2020. Chess Pieces: Board Setup, Movement, And Notation. [online] Available at: <https://www.thechesswebsite.com/chess-pieces/> [Accessed 19 July 2020].
- Plaat, A., 1995. A Minimax Algorithm Better Than SSS*?. Edmonton: Dept. of Computing Science, University of Alberta.
- Pollack, M. E., 1995. Artificial Intelligence -- A Modern Approach -- A Review. AI Magazine,
- En.wikipedia.org. 2020. Rules Of Chess. [online] Available at: <https://en.wikipedia.org/wiki/Rules_of_chess> [Accessed 19 July 2020].