SRM
INSTITUTE OF SCIENCE & TECHNOLOGY

COLLEGE OF ENGINEERING & TECHNOLOGY
SRM INSTITUTE OF SCIENCE & TECHNOLOGY
S.R.M. NAGAR, KATTANKULATHUR – 603 203

BONAFIDE CERTIFICATE

Certified that this project report **"GAME OF CHESS"** is the bonafide work of **"SRINATH M (RA2011003010176), AKHIL RAJESH (RA2011003010176), SHRIDHARSHAN V A K (RA2011003010180)"** who carried out the minor project under my supervision. Certified further, that to the best of my knowledge, the work reported herein does not form any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

Department of Computing Technologies
Kattankulathur
603 203
SRM IST

12/5/23

SIGNATURE

M. Rajalakshmi
GUIDE
Assistant Professor
Department of Computing Technologies

M. Pushpalatha

SIGNATURE

Dr. M. Pushpalatha
HEAD OF THE DEPARTMENT
Professor & Head
Department of Computing Technologies

GAME OF CHESS

A MINI PROJECT REPORT

18CSC305J - ARTIFICIAL INTELLIGENCE

Submitted by

SRINATH M [RA2011003010176]
SHRIDHARSHAN VA K
[RA2011003010180]
AKHIL RAJESH[RA2011003010174]

Under the auidance of
M. Rajalakshmi
Assistant Professor
Department of Computing Technologies

In partial satisfaction of the requirements for the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE & ENGINEERING

of

FACULTY OF ENGINEERING AND TECHNOLOGY

SCHOOL OF COMPUTING

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR - 603203

MAY 2023

COLLEGE OF ENGINEERING & TECHNOLOGY
SRM INSTITUTE OF SCIENCE & TECHNOLOGY
S.R.M. NAGAR, KATTANKULATHUR – 603 203

BONAFIDE CERTIFICATE

Certified that this project report **"GAME OF CHESS"** is the bonafide work of **"SRINATH M (RA2011003010176), AKHIL RAJESH (RA2011003010176), SHRIDHARSHAN V A K (RA2011003010180)"** who carried out the minor project under my supervision. Certified further, that to the best of my knowledge, the work reported herein does not form any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

SIGNATURE

M. Rajalakshmi
GUIDE
Assistant Professor
Department of Computing Technologies

SIGNATURE

Dr. M. Pushpalatha
HEAD OF THE DEPARTMENT
Professor & Head
Department of Computing Technologies

ABSTRACT

In recent years, computer games have become a common form of entertainment. Fast advancements in computer technology and internet speed have helped entertainment software developers to create graphical games that keep a variety of players' interest. The emergence of artificial intelligence systems has evolved computer gaming technology in new and profound ways. Artificial intelligence provides the illusion of intelligence in the behavior of NPCs (Non-Playable-Characters). NPCs are able to use the increased CPU, GPU, RAM, Storage, and other bandwidth-related capabilities, resulting in very difficult gameplay for the end user. In many cases, computer abilities must be toned down in order to give the human player a competitive chance in the game. This improves the human player's perception of fair gameplay and allows for continued interest in playing. A proper adaptive learning mechanism is required to further this human player's motivation. During this project, past achievements of adaptive learning on computer chess gameplay are reviewed and adaptive learning mechanisms in computer chess gameplay are proposed. Adaptive learning is used to adapt the game's difficulty level to the players' skill levels. This adaptation is done using the player's game history and current performance. The adaptive chess game is implemented through the open-source chess game engine Beowulf, which is freely available for download on the internet.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

AES	Advanced Encryption Standard
ANN	Artificial Neural Network
CSS	Cascading Style Sheet
CV	Computer Vision
DB	Data Base
DNA	Deoxyribo Neucleic Acid
SQL	Structured Query Language
SVM	Support Vector Machine
UI	User Interface

CHAPTER 1

INTRODUCTION

1.1 HISTORY OF COMPUTER CHESS PLAY

1.1.1 An “Elaborate Hoax”

The first chess playing automaton was built in 1769 by the Hungarian-born engineer Baron Wolfgang for the amusement of the Austrian Queen Maria Theresa.[3] Later, it was revealed that it was a hoax and its outstanding capability originated from a human player that was hidden inside it. It was later described in an essay by Edgar Allan Poe, "Maelzel's Chess-Player.". The first article on computer chess was “Programming a computer for playing chess " published in *Philosophical Magazine*, March 1950 by Claude Shannon, a research worker at Bell Telephone Laboratories in New Jersey. In his article, Shannon described how to program a computer to play chess based on position scoring and move selection.

The first computer chess program was written by Alan Turing in 1950.[3] This was soon after the second World War. At that time the computer was not invented so he had to run the program using pencil and paper and acting as a human CPU and each move took him between 15 to 30 minutes. He also proposed the Turing test, which stated that in time a computer can be programmed to acquire abilities that need human intelligence (like playing a chess game). If the human who is playing does not see the other human or

computer during the game, he would not know whether he is playing with a human or with a computer.

Later in 1951, Turing wrote his program named “Turbochamp" on the Ferranti Mark I computer at Manchester University.[3] He never completed his program but his colleague, Dr. Dietrich Prinz wrote a chess playing computer program for the Ferranti computer that was able to examine every possible moveuntil it found the optimal move. An early computer was built under the direction of Nicholas Metropolis at the Los Alamos Scientific Laboratory.[3] It was named MANIAC I and it was based on the von Neumann architecture of the IAS. This machine was programmable, filled with thousands of vacuum tubes and switches, and it was able to execute 10,000 instructions per second. MANIAC I was able to play chess using a 6"x6" chessboard and it took twelve minutes for itto search four moves ahead.

The first chess program that played a chess game professionally was created by Alex Bernstein, an IBM employee in 1957.[3] He and his three colleagues created a chess program at the Masschusetts Institute of Technologythat ran on an IBM 704. It took about eight minutes for their chess program to make a move. In 1958, the first chess program to be written in a high-level language was developed by Allen Newell, Herbert Simon and Cliff Shaw at Carnegie-Mellon.[3] Their program was called NSS (Newell, Simon, Shaw) and ittook about an hour to make a move. They combined algorithms that searched for good moves with heuristics algorithms and ran on a JOHNNIAC computer.

1.1.2. Artificial Intelligence in Computer Chess

The first chess program that played chess credibly was written by Alan Kotok (1942-2006) at the Massachusetts Institute of Technology.[3] It was writtenon an IBM 7090 and it was able to beat chess beginners.

Richard Greenbaltt, an MIT expert in artificial intelligence, with Donald Eastlake wrote their chess program in 1960.[3] Their chess program was named MacKack and it was the first chess program to play in human tournaments. It wasthe first to be granted a chess rating and draw and win against a human being in tournament play.

1.1.3. Sargon Computer Chess

A chess game named SARGON was written by Dan and Kathleen `Kathe' Spracklen on a Z80- based computer called *Wavemate Jupiter III* using assembly language.[3] It was introduced in 1978 at the West Coast Computer Fair and it won the first computer chess tournament held for microcomputers. Its name “Sargon" was taken from the historical kings, Sargon of Akkad or Sargon of Assyria, and it was written entirely in capitals because early computer operating systems like CP/M did not support lower-case file names.

Three doctoral students created the chess program Chiptest in 1985.
Their chess program was developed into Deep Thought which shared first place with Grandmaster Tony Miles in the 1988 U.S. Open championship and defeated the sixteen year-old Grandmaster Judit Polgar in 1993. [3]

1.1.4. Deep Blue

The world champion Gary Kasparov was defeated by IBM's chess program, Deep Blue in a six-game series.[3] Deep Blue was designed to consider several billion possibilities at once and it used a series of complicated formulas that took into consideration the state of the game. The game also kept a record of several past matches and Kasparov found this out which is why the game was able to beat him.

1.2 COMPONENTS OF COMPUTER CHESS PLAY

1.2.1. Chess Engine

Chess engine is a computer program that decides what move to make during the game. It makes some calculations based on the current position to decide the next move.[14] There are many chess engines available to download but in this project we will use Beowulf chess engine that is open source. Some other open source chess engines include: stockfish, Gull, Protector, Minkochess, Texel, Scorpio, Crafty, Arasan, Exchess, Octochess, Rodent, Redqueen, and Danasah. There are also other chess engines that are free to use like: Critter, Hannibal, Spike, Quazar, Nemo, Dirty, Gaviota, Prodeo and Nebula. Some commercial chess games include: Houdini, Rybka, Komodo, Vitruvius, Hiarcs, Chiron, Shredder, and Junior. [5]

1.2.2. GUIs for Chess Games

GUI (Graphical User Interface) is a user interface where the user and the chess game can interact with each other. [15] Unlike text-based user interfaces

where input and output is in plain text, GUI uses a more complicated graphical representation of the program and also it creates a more flexible user interaction by using pointing devices, mice, pens, and graphics tablets that allow the user to interact with the computer more easily. A graphical interface for the chess game has a graphical chess board that allows users to enter moves by clicking on the board or dragging a piece on the board just like a real chess game. [15]

There are many chess GUIs available to download. Some chess GUIs include Aquarium, Arena, Chess Academy, Chess for Android, ChessGUI, ChessPartner GUI, Chess Wizard, ChessX, Fritz GUI, Glaurung GUI, Hiarcs, Chess Explorer, jose, Mayura Chess Board, Scid vs. PC, Shredder GUI, Tarrasch GUI, WinBoard, and XBoard.

CHAPTER 2

METHODOLOGY

2.1 GAME TREE AND CHESS

In chess, game tree is a directed graph whose nodes are positions in the game and edges are moves the players make. Each node in the game tree has avalue and leaf node that the game ends at are labeled with the payoff earned by each player.[4] In the game tree, the player's position in the game is represented by nodes and the edges represent the moves they can make at that position. The Beowulf chess engine finds the best move in game by searching the game tree. The Game tree is also important in artificial intelligence because the search algorithm, using the MinMax algorithm or other search algorithms, searches the game tree to find the best move. The complete game tree starts from the initial position and contains all the possible moves in the game. In a complete game tree, the number of leaf nodes is the number of possible different ways the game can be played. Some game trees like tic-tac-toe are easier to search but searching the complete tree in larger games like chess takes a longer time. Instead of searching the complete tree, the chess program searches a partial game tree. It starts from the current position and it searches as many plies as it can in the limited time. Increasing the search depth (number of plies it searches) will result in finding a better move but it takes more time to search

2.1 SEARCH ALGORITHMS

In computer chess games, the search algorithms are used to find the best move. The search algorithm will look ahead for different moves and evaluate the positions after making each move. Different chess engines use different search algorithms. Some search algorithms include: Minimax algorithm, NegaMax algorithm, NegaScout algorithm, and Alpha-Beta algorithm.

2.2.1. Minimax Algorithm

The score in a two player zero-sum game like chess can be determined bythe Minimax algorithm after a certain number of moves. In the Minimax algorithm we are trying to minimize the possible loss in a worst case scenario. We can alsothink of it as maximizing the minimum gain.[7] The Minimax theorem states that for every 2-person zero-sum game with finite strategies, there exists a value V, such that given player 2's strategy, the best payoff possible for player 1 is V and given player 1's strategy, the best payoff possible for player 2 is -V. This means that player 1 strategy brings a best payoff possible of V for him regardless of player 2's strategy and player 2 strategy brings a best payoff possible of -V for him regardless of player 1's strategy. This theorem was established by John von Neumann. He says, "As far as I can see, there could be no theory of games ... without that theorem ... I thought there was nothing worth publishing until the

{Minimax Theorem} was proved.[7]

2.2.2. Alpha-Beta Pruning

The Alpha-Beta pruning algorithm is an enhancement to the Minimax search algorithm.[8] It applies a branch and bound technique to eliminate the need to search large portions of the game tree. When it finds at least one possibility that proves the move is worse than a previously examined move, it stops completely evaluating that move and those moves will not be evaluated in the future. If we apply the Alpha-Beta algorithm to the Minimax tree it will return the same move the Minimax algorithm would return but it prunes away portions of the game tree that possibly can't influence the decision. The values of Alpha and Beta represent the minimum score that the maximizing player is assured and the maximum score that the minimizing player is assured, respectively.[8]

2.2.3. NegaScout

The Negascout is a search algorithm that computes the Minimax value of a node in a game tree faster than Alpha-Beta pruning.[9] The Nega-Scout algorithm works faster than Alpha-Beta pruning because it doesn't examine nodes that can be pruned by Alpha-Beta. The NegaScout needs a good move ordering in order to work. If the game has random move ordering then the Alpha-Beta algorithm still works best. The NegaScout search algorithm was invented by Alexander Reinefeld and it gives typically 10 percent performance increase in chess engines compared to alpha-beta pruning.[9]

2.2.4. NegaMax

The Negamax search is a variant form of the Minimax algorithm. It simplifies the Minimax algorithm based on the fact that:

max (a, b) = -min (-a, -b).[10]

This means that a value that a position has for one player is the negation of the value that position has for the other player. This lets us use a single procedure to value both positions. It is different from NegaScout which uses the alpha-beta pruning which is an enhancement to the MiniMax algorithm.[10]

2.3 BOARD REPRESENTATIONS

2.3.1. Piece Lists

Since the early computers had a very limited amount of memory, spending 64 memory locations for the pieces was too much. Instead of that, early chess games saved the locations of up to 16 pieces in their memory. In newer chess games, the piece list is still in use, but they also use a separate board representation structure. This allows the chess engine to access the pieces faster.

2.3.2. Array Based

The chess board can also be represented by creating an 8x8 two- dimensional array or one 64 element one-dimensional array. Each one of these 64 elements stores the information for each piece on the chess board. One approach is to use 0 for an empty square, positive numbers for white pieces and negative numbers for black pieces. For example, the white king can be +4 and

the black king -4. In this approach the chess game has to check each move to besure it is on the board. This will slow down the chess game and decrease performance. To solve this problem we can use a 12x12 array and assign the value 99 to spaces on the edge of the board, where pieces cannot be placed.

This will let the chess game know that the destination square is not on the board while making the moves. For better memory usage, we can use a 10x12 array. This representation will have the same functionality but the leftmost and the rightmost edges are overlapped. In some other chess games, 16x16 arrays are used. This allows the programmers to achieve better performance and implementsome coding tricks due to the increased availability of memory (for example, attacks).

2.3.3. 0x88 Method

In the 0x88 method, instead of a 64 bit array, a one-dimensional 16x8 array is used. There are actually two boards next to each other and the board on the left has the actual values. For each square on the board, binary layout 0rrr0fff is used to coordinate rank and file in the array (rrr are the 3 bits that represent the rank and fff are 3 bits that represent the file). We can check to see if a destination square is on the board by ANDing the square value with 0x88 (10001000 in binary). If the result of AND is not zero that means the square is offboard. To know if two squares are in the same row, column, or diagonal, we can subtract the values of two square coordinates.

2.3.4. Bitboard

Another way to represent the chess board is using Bitboard. In this method we use 64-bits to save the states of each place on the board, a sequence of 64 bits in which each one can be true or false. We can also use a series of Bitboards to represent the chess board. This allows computers with 64-bit processors to use bit parallel operations and to take advantage of their increased processing power.

CHAPTER 3

DESIGN AND IMPLEMENTATION

3.1 THE BEOWULF CHESS ENGINE AND DIFFICULTY LEVEL

3.1.1. The Beowulf Chess Engine

In this project we made changes to the Beowulf chess engine. The goal ofthe Beowulf chess project was to create a challenging chess game that is open source, is freely available for download, and is well documented. It can be downloaded from: http://www.frayn.net/beowulf/

This chess engine works in text mode but it can be integrated with graphical interfaces like Xboard and Windboard. It's written in C language and it consists ofthe following files:

Main.c

This is the main file for the chess game. It includes the main structure forthe program, loads the data into the program, and initializes and runs the state machine. This is the main file that we will modify during this project.

Eval.c

This file is used for evaluating board positions. It consists of different functions, such as Analyse, which is used to display and print the current board analysis and Eval function, which evaluates game position.

Beowulf.cfg

This is the configuration file for the Beowulf chess engine. It gives a default skill level of 1 to both Black and White players and loads the opening book and the personality file. It also turns on RESIGN, which lets Beowulf resign in a losing position.

Pers.c

This file contains the personality code. It was disabled during the project to change the difficulty level of the game.

Parser.c

This file is used to parse the input string from the player. The input that the user enters is passed to the Parseinput function which interprets the input.

Board.c

This file contains all the algorithms working with Bitboards.

Checks.c

This file has all the functions that check for checkmates, threats, and checks.

Moves.c

For each position in the game, a list of all possible moves is created using the functions in this file.

Tactics.c

This file contains different functions that evaluate the value for different pieces in different board positions.

3.1.2. Game Skill Levels

In this game, skill level is a number from 1 to 10. 1 is the lowest skill level and 10 is the highest. The higher the skill level, the stronger the player is. During the game the skill level for the White or Black player can be changed by assigning a number from 1 to 10 to their skill level. The skill level for the Black player can be changed by assigning a number from 1 to 10 to Params.BSkill.

Also, we can change the white player's skill level by assigning a number from 1 to 10 to Params.WSkill.

Another way to define the skill level is by giving the depth parameter a value in comp function. Since the chess engine uses the comp function to calculate the best move, we pass a parameter to it in order to define the difficulty level. The White player uses comp function and we pass parameter "i" to it. The Black player uses comp2 and we pass parameter "j" to it. This parameter can be changed during the game to change the difficulty level of that player.

3.1.3. Adding New Function to Evaluate Movements

The Beowulf game by default starts in player mode. It shows a prompt for the user to enter his move. By entering the Xboard command, the computer

starts playing. The chess engine uses comp function in comp.c to calculate the best move. In order to create an adaptive chess engine I created another function named comp2. By checking the Current_Board.side we can know which side is taking its turn. This variable can be either white or black depending on which player is taking their turn. In order to have an evaluation of every movement we created the Analyse2(Current_Board) function in parser.c. This function compares the black player's points to the white player's points. If both players have the same number of points it returns 0, if the Black player is ahead,then it will subtract the White player's score from the Black player's and return that value. If the White player is ahead, then it will return the difference as a negative value. Each time after the White player makes his move, we call the evaluation function to see the player's position in the game. The Analyse2 function is shown below:

```
int Analyse2(Board
B) {int side = B.side,
sc; double score;
BOOL TB = FALSE;
CompDat Params;
fprintf(stdout,"Current Position\n ---------- \n\n");
fprintf(stdout,"White Points:
%d\n",B.WPts);fprintf(stdout,"Black Points:
%d\n",B.BPts); if (B.WPts==B.BPts)
        {
        fprintf(stdout,"(Even
        Sides");return(0);
```

```
    }
  if (B.WPts<B.BPts) {
    fprintf(stdout,"(Black is Ahead by %d Pt",B.BPts-B.WPts);
    if (B.BPts-B.WPts>1) fprintf(stdout,"s");
    return(B.BPts-B.WPts);
  }
  if (B.WPts>B.BPts) {
    fprintf(stdout,"(White is Ahead by %d Pt",B.WPts-B.BPts);
    if (B.WPts-B.BPts>1) fprintf(stdout,"s");
          return(-(B.WPts-B.BPts));
  }
  fprintf(stdout,")\n");
}
```

3.1.4. Giving the Computer Player a Difficulty Level

In order to change the difficulty level of a player, we can change the valueof Params.Time, Params.MoveTime, or Params.Depth parameters.

Params.MoveTime is the maximum time for a move in seconds, and Params.Depth is the minimum search depth in ply. Depth overrides the Params.Movetime parameter.

Also, turning off the opening book by disabling the LoadOpeningBook function in the main function and/or turning off the personality file by disabling the LoadPersonalityFile function will decrease the difficulty level of Beowulf chess game.

The following parameters are defined in the Defaults function and are set to values each time the game starts in the main function. In order to change the computer player level we can change these parameters.

```
  /* Setup the computer parameters */
  Params.Depth = 5;        /* Minimum Search Depth in ply. Overrides 'MoveTime' */
Params.Time = Params.OTime = 1; /* Total Clock Time = 5 minutes in centiseconds
*/
  Params.MoveTime = 1; /* Maximum time in seconds. 'Depth' overrides this */
  Params.Test = FALSE;      /* Not running a test suite */
  Params.Mps = 0;           /* Moves per session. 0 = all
  */Params.Inc = 0;           /* Time increase per move */
  automoves = 0;
  NHash = 0;
  TableW = TableB =
  NULL;Randomise();
  GlobalAlpha = -
  CMSCORE;GlobalBeta =
  CMSCORE;
}
```

During the game we can change the difficulty level of the player by passing the depth value to them. Since we use comp function for the White player and comp2 for the Black player we pass parameters “i” and “j” to them. These parameters can change during the game to create the adaptive engine.This is the comp function for the White player: [13]

```
MOVE Comp(int i) {
  int
depth=i,inchk,val=0,score=0;
#ifdef BEOSERVER
  int n;
```

```
  float
ExtendCostAv;
#endif //
BEOSERVER
  longlong LastPlyNodecount=1;
  BOOL Continue=FALSE,resign=FALSE;
  Board BackupBoard=Current_Board,*B = &BackupBoard;
  MOVE Previous=NO_MOVE,BookMove=NO_MOVE,BestMove;
  HashElt *Entry=NULL;
  BOOL bBreakout = FALSE;

   /* Reset the input flag before we do anything.  This flag tells us about how
   *  and why the comp() procedure exited.  Normally it is INPUT_NULL,
    * which tells * us all is OK.  Sometimes it is set to different values, often in
analysis mode */
  InputFlag = INPUT_NULL;


   /* Check the Opening Book First */
  if (AnalyseMode==FALSE &&
    BookON) {BookMove =
    CheckOpening(B,&val);
  }
  if (BookMove!=NO_MOVE) {
     /* Check some values for the book move, i.e. EP and castle */
    BookMove = CheckMove(BookMove);
    if (UCI) fprintf(stdout,"bestmove ");
    else if (XBoard) fprintf(stdout,"move
    ");
#ifdef BEOSERVER
    else fprintf(stdout,"Best Move =
");#endif
    PrintMove(BookMove,TRUE,stdout);
    #ifdef BEOSERVER
```

```
    fprintf(stdout,"\
n");#endif
    fprintf(stdout,"  <Book %d%%>\n",val);
    if (!AnalyseMode && (XBoard || AutoMove)) {
      MoveHistory[mvno] = BookMove;
      UndoHistory[mvno] = DoMove(&Current_Board,BookMove);
      mvno++;
    }
    AnalyseMode =
    FALSE;return
    BookMove;
  }

   /* Setup the Hash Table
  */SetupHash();

 /* Setup the draw by repetition check. InitFifty holds the number of moves
*backwards we can look before the last move which breaks a fifty move draw
*chain. We need to store this to help with the draw checking later on. */
  InitFifty = GetRepeatedPositions();

   /* Reset Initial values
  */ResetValues(B);

   /* Test for check */
  inchk = InCheck(B,B->side);

   /* Display the current positional score */
  fprintf(stdout,"Current Position = %.2f\n",(float)InitialScore/100.0f);
  position=InitialScore;  //We add this to store the position for the white player
```

```
  /* Count the possible moves */ TopPlyNMoves
  = (int)CountMoves(B,1,1);
  if (!XBoard) fprintf(stdout,"Number of Possible Moves =
%d\n\n",TopPlyNMoves);if
  (TopPlyNMoves==0) {
            //If it's end of game
            fprintf(stdout,"Game Ended!\n");
          /*  fprintf(stdout,"Black player level after adoption is:
          %d\n",adoptionlevel);*/
          /*  getchar();*/

    if (inchk)
                  {
                  fprintf(stdout,"You are in Checkmate!\n");
                  return NO_MOVE;
                  }
           else
                  {
                  fprintf(stdout,"You are in Stalemate!\n");
                  return NO_MOVE;
                  }
                  return NO_MOVE;
    if (AnalyseMode) bBreakout = TRUE;
    else return NO_MOVE;
  }

#ifdef BEOSERVER
  // Reset the NodeID count
```

```
  NextNodeID = 0;
  fprintf(stdout,"Calculating Approximate NODE Complexities\n");
  // Calculate Node complexities for various depths
  for (n=1;n<5;n++) {
    // Output the details
  fprintf(stdout,"Count Nodes : Depth = %d,  N=%d\n",n,(int)CountMoves(B,n,1));
  }
#endif
   /* Generate the hash key for this position */
  GenerateHashKey(B);
   /* Start the Game Clock
  */SetStartTime();
   /* Set up the necessary algorithm variables */
  RootAlpha = GlobalAlpha; RootBeta = GlobalBeta;
  TBHit = TRUE;
#ifdef BEOSERVER
  /* Set up the Node Table for the Parallel algorithm */
  NodeTable = (NODETABLE
*)calloc(sizeof(NODETABLE),NODE_TABLE_SIZE);
  for (n=0;n<NODE_TABLE_SIZE;n++) NodeTable[n].entries = 0;
  /* Set up the minimum parallel depth to 'not defined' */
  SeqDepth= -1;
#endif // BEOSERVER

                /*   ---===   Begin Iterative Deepening Loop       ===------*/
  do {
     // break out if we're in analysis mode and this position is
     // either checkmate or stalemate.  Go straight into a loop
     // waiting for input instead
```

```
    if (bBreakout)

    break; GlobalDepth

    = depth;

     /*   --==   Do Search Recursion Here        ==--    */

#ifdef BEOSERVER
     /* If we've spent long enough searching natively then start
      * distributing nodes to the peer nodes.  If we're pondering
      * then don't bother with the parallel search - just fill up
      * the native hash tables.  Don't parallel search easy nodes. */
    if (bParallel && GetElapsedTime() > MIN_SEQ_TIME && !Pondering &&
!bEasyNode) {
      if (SeqDepth == -
        1) {SeqDepth =
        depth;
        // Estimate branching complexity
        ExtendCost     = (float)Nodecount / (float)TopPlyNMoves;
        ExtendCostAv = (float)pow(ExtendCost,1.0f/(float)(depth-1));
        fprintf(stdout,"Averaged Extend Cost = %.3f\n",ExtendCostAv);
        ExtendCost     = (float)Nodecount / (float)LastPlyNodecount;
        fprintf(stdout,"Last Ply Extend Cost = %.3f\n",ExtendCost);
        ExtendCost = min(ExtendCost, ExtendCostAv);
        fprintf(stdout,"Adopted Minimum Value of %.3f\n",ExtendCost);
      }
      // Do the parallel search
      score = RunParallel(B,depth,inchk,InitFifty,GetElapsedTime());
    }
    else
#endif // BEOSERVER
```

```
          * table! */
         if (!AbortFlag && (score <= RootAlpha || score >= RootBeta)) {
           RootBeta = CMSCORE;
           RootAlpha = -
           CMSCORE;score =
Search(B,RootAlpha,RootBeta,depth*ONEPLY,0,inchk,InitFifty,0,NO_MOVE);
         }
        }
        else if (!AbortFlag && IsCM(score) && Post) PrintThinking(score,B);
      }
      else if (!AbortFlag) PrintedPV = FALSE;

           // Is this an easy node? (i.e. a simple recapture move that must be
played)
      if (TopOrderScore > NextBestOrderScore + EASY_MOVE_MARGIN &&
BestMoveRet != NO_MOVE && TopOrderScore ==
SEE(B,MFrom(BestMoveRet),MTo(BestMoveRet),IsPromote(BestMoveRet)) *
100) {
                  bEasyMove = TRUE;
           }
           else bEasyMove = FALSE;
    }

           /* --==  Search Finished ==--  */

           /* Probe the hashtable for the suggested best move */
    Entry = HashProbe(B);
    if (Entry) {
      BestMove = Entry->move;
```

```
      score = (int)Entry->score;
    }
    else {BestMove = NO_MOVE;fprintf(stdout,"Could Not Find First Ply Hash
Entry!\n");}
    if (BestMove == NO_MOVE) {fprintf(stderr,"No Best Move! Assigning
previous\n");BestMove = Previous;}

     /* If we aborted the search before any score was returned, then reset to the
      * previous ply's move score */
    if (AbortFlag && BestMove == NO_MOVE) score = PreviousScore;
     /* Otherwise store what we've found
    */else {
      PreviousScore =
      score;Previous =
      BestMove;
    }
    BestMoveRet = Previous;

     /* Go to the next depth in our iterative deepening loop */
    depth++;

     /* Check to see if we should continue the search */if
    (AnalyseMode) Continue = Not(AbortFlag);
    else Continue = ContinueSearch(score,B->side,depth);

  } while (IsCM(score)==0 && (TopPlyNMoves>1 || AnalyseMode) && Continue
&& !TBHit);

                /*   --==  End Iterative Deepening Loop  ==--        */
```

```
  /* Store total time taken in centiseconds */
 TimeTaken = GetElapsedTime();

  /* Expire the hash tables if we're playing a game. This involves making all the
   * existing entries 'stale' so that they will be replaced immediately in future,
   * but can still be read OK for the time-being. */
 if (!Params.Test && !AnalyseMode && (Params.Time > 100)) ExpireHash();

  /* If we've not got round to printing off a PV yet (i.e. we've had an early exit
   * before PrintThinking() has been called) then do so now. */if
 (!PrintedPV) {PrintedPV=TRUE;PrintThinking(score,B);}

  /* Check to see if we should resign from this position.  Only do this if we'vethought
   * for long enough so that the search is reliable and the current position is also
   * losing by at least a pawn, depending on the skill level. */
 if (!Pondering && !Params.Test && Params.Resign && depth>=7 &&
Params.Time<Params.OTime && InitialScore<(Skill*10)-200)
    resign = CheckResign(score);

  /* Check to see if we should offer a draw from EGTB search */
#ifdef OFFER_TB_DRAW
 if (!Pondering && !Params.Test && XBoard && TBON
&&ProbeEGTB(B,&score,0)) {
   if (!IsCM(score)) fprintf(stdout,"offer draw\n");
 }
#endif

  /* Write the PV to a text string in case we're writing a logfile */
```

```
  WritePVToText(B);

   /* Print timing and search information
  */if (!XBoard) PrintInfo(score);

   /* Output details of the move chosen, and play that move if necessary */if
  (InputFlag == INPUT_RESIGN) InputFlag = INPUT_STOP;
  if (UCI) {
    fprintf(stdout,"bestmove ");
    PrintMove(Previous,FALSE,stdout);
    fprintf(stdout,"\n");
  }
  if (!UCI && !Pondering && (AutoMove || XBoard) && !resign &&
!AnalyseMode&&
       InputFlag != INPUT_STOP)
    {if (XBoard) {
    fprintf(stdout,"move ");
       PrintMove(Previous,FALSE,stdout);
       fprintf(stdout,"\n");
    }
    MoveHistory[mvno] = Previous;
    UndoHistory[mvno] = DoMove(&Current_Board,Previous);
    mvno++;
    MoveHistory[mvno] = NO_MOVE;
     /* We've been told to move immediately, we've just moved as required,
      * so flag that this is accomplished */
    if (InputFlag == INPUT_MOVE_NOW) InputFlag = INPUT_NULL;
  }
```

```
  /* If we're altering the position whilst in analysis mode, then do so */if
 (!Pondering && AnalyseMode) {
   // We got a CM score from this analysis - just wait for more commandsif
  (InputFlag == INPUT_NULL) {
    // Wait for something to do
    while ((InputFlag = CheckUserInput()) == INPUT_NULL);
  }
  if (InputFlag == INPUT_MOVE) {
    MoveHistory[mvno] =
    MoveToPlay;
    UndoHistory[mvno] = DoMove(&Current_Board,MoveToPlay);
    mvno++;
    MoveHistory[mvno] = NO_MOVE;
  }
  if (InputFlag ==
    INPUT_UNDO) {mvno--;
    UndoMove(&Current_Board,MoveHistory[mvno],UndoHistory[mvno]);
  }
  if (InputFlag == INPUT_NEW) {
    ResetBoard(&Current_Board);
  }
 }

  /* Tidy up */
 if (InputFlag == INPUT_NULL || InputFlag == INPUT_STOP || InputFlag ==
INPUT_WORK_DONE) AnalyseMode=FALSE;
 if (IsCM(score)==1) CMFound=TRUE;
 if (Params.Test || AnalyseMode) ResetHash();
#ifdef BEOSERVER
 for (n=0;n<NODE_TABLE_SIZE;n++) {
```

```
    if (NodeTable[n].entries > 0) free(NodeTable[n].list);
   }
  free(NodeTable);
#endif //
BEOSERVER

   /* Return the best move that we found */
  return Previous;
}
```

The Black player uses Comp2 function to make moves. Comp2 function isthe same as Comp function but takes a “j” parameter. This parameter is assignedto depth.

3.1 MAKING A COMPUTER PLAY AGAINST A COMPUTER

By default, the Beowulf chess engine starts playing in user player mode (human player vs computer player). We made some changes to the code to make the computer play against itself. We created strings input2, input3 and input4. Input2 and input4 are assigned to “comp\n”. Input 3 is assigned to “xboard\n” and input5 is assigned to “eval\n”. By calling the ParseInput function with parameter input4 the computer starts playing the game (as the white player). When it’s time for the black player to make a move we call ParseInput with parameter input2.

In main function we define the skill level for the black player and white player with the following commands:

Params.BSkill=1;

Params.WSkill=5;

When the ParsInput function is called with ”comp\n” value it returns “S_SEARCH” which has a value of 1 and sets the computer processing. We have created another copy of comp() function in comp.c and we named it comp2().The following code will use the comp() function for the white player and the comp2() function for the black player:

```
If (Current_Board.side==WHITE) { if (Comp()==-1) ComputerGO = FALSE;}
else
{if (Comp2()== -1) ComputerGO = FALSE;}
```

Current_Board.side value lets us know which player is going to make a move. Based on its value, 0 for the white player and 1 for the black player, wecall different functions. This makes the computer move for the black player.

3.2 Current Position

The approximations we are using to calculate the player’s position in the game are:

0.3 shows that the player has first move advantage.

0.6 +- shows a slight advantage for the player

0.9 +- shows a clear advantage for the player

Any number 1.3 + shows the player is winning the game.

If the current position is a number between 0 and 1.3, the game is still close. When the current position gets to a number more than 1.3 the player hasclear advantage and he is winning the game. If the current position is greater

than 1.3 for 3 moves, then we will decrease the skill level. The game will continueon. Each time the player's current position is greater than 1.3 for three consecutive moves, we decrease the skill level. This continues until two players have the same difficulty level.

3.3 CREATING THE ADAPTIVE CHESS ENGINE

During the game we check the current position for each player to see if that player is playing better than his opponent and use this to adapt the player's level as explained above. Current position for each player shows if that player is winning the game or if he is losing. Any positive number from 0 to 1 shows advantage for that player and any number greater than 1.3 shows that player is winning. We created a global variable named position to save the current positionfor each player during the game.

```
/* Display the current positional score */
  fprintf (stdout,"Current Position = %.2f\n",(float)InitialScore/100.0f);
  position=InitialScore;  //We add this to store the position for the white player
```

In the main function we can check that value to see if that player has advantage or not:

```
if ((float)position/100.0f>0.9) { fprintf(stdout,"White player has advantage"); }
```

We created two variables named bposition and wposition and gave themeach a value of 0. Each time one player has advantage we add 1 to these variables:

```
if ((float)position/100.0f > 1.3)
```

```
{
bposition++;
if (bposition>=3 && wdifficulty<5)
                    {
                    fprintf(stdout,"changing white player level");
                    wdifficulty++;
                    Params.WSkill+
                    +;bposition=0;
                    }
}
}
```

The following code increases the difficulty level of the white player when the black player has had clear advantages for 3 moves. This lets the white playerto adapt to the black player during the game.

3.4 Saving the Skill Level

After each game, the white player keeps the adapted skill level and starts the next game with that skill level. If the comp function returns -1 that means thatthe game is over. This is the condition that shows us the game ended after the white player moved:

```
if (Comp(wdifficulty) == -1)          {//if the game ends
```

And we use this condition to check to see if the game ended after the black player moved:

```
if (Comp2(5) == -1)                   {//if the game ends
```

We created a label named “play” to start the next game. After each game ends we call the functions to initialize the game and then start the new game by calling that label.

In order to save the output to a file we used the following command in the command prompt:

Beowulf >> output.txt

This command redirects the output to a text file named output.txt in the project folder.

We can open this file later using a text editor like notepad to see the history ofthe game play in order to see the adaptive process.

CHAPTER 4

TESTING AND RESULTS

4.1. Testing Cases

We have three test cases for the adaptive engine. In the first test case the computer player (white player) plays with another computer player (black player).The black player has a skill level of 5 and the white player starts with a lower skill level and adapts to the black player during the game. In the second test case, the black player starts the game with a skill level of 1 and the white player starts the game with a higher skill level. During the game the white player adapts to the black player's skill level. In the 3th test case, the computer player (black player) starts the game with a skill level of 5 and adapts to the human player (white player) during the game.

CODE

```
    inport itertools


WHITE = "white"
BLACK = "black"
```

```
class Game:
    #ive decided since the number of pieces is capped but the type of pieces is not (pawn transformations), I've already coded much of the modularity to support just using a dictionary of pieces
    def __init__(self):
        self.playersturn = BLACK
        self.message = "this is where prompts will go"
        self.gameboard = {}
        self.placePieces()
        print("chess program. enter moves in algebraic notation separated by space")
        self.main()



    def placePieces(self):


        for i in range(0,8):
            self.gameboard[(i,1)] = Pawn(WHITE,uniDict[WHITE][Pawn],1)
            self.gameboard[(i,6)] = Pawn(BLACK,uniDict[BLACK][Pawn],-1)

        placers = [Rook,Knight,Bishop,Queen,King,Bishop,Knight,Rook]

        for i in range(0,8):
            self.gameboard[(i,0)] = placers[i](WHITE,uniDict[WHITE][placers[i]])
            self.gameboard[((7-i),7)] = placers[i](BLACK,uniDict[BLACK][placers[i]])
        placers.reverse()



    def main(self):

        while True:
            self.printBoard()
            print(self.message)
            self.message = ""
            startpos,endpos = self.parseInput()
            try:
                target = self.gameboard[startpos]
            except:
                self.message = "could not find piece; index probably out of range"
                target = None

            if target:
```

```
            print("found "+str(target))
            if target.Color != self.playersturn:
                self.message = "you aren't allowed to move that piece this turn"
                continue
            if target.isValid(startpos,endpos,target.Color,self.gameboard):
                self.message = "that is a valid move"
                self.gameboard[endpos] = self.gameboard[startpos]
                del self.gameboard[startpos]
                self.isCheck()
                if self.playersturn == BLACK:
                    self.playersturn = WHITE
                else : self.playersturn = BLACK
            else :
                self.message = "invalid move" +
str(target.availableMoves(startpos[0],startpos[1],self.gameboard))
                print(target.availableMoves(startpos[0],startpos[1],self.gameboard))
        else : self.message = "there is no piece in that space"

    def isCheck(self):
        #ascertain where the kings are, check all pieces of opposing color against those
kings, then if either get hit, check if its checkmate
        king = King
        kingDict = {}
        pieceDict = {BLACK : [], WHITE : []}
        for position,piece in self.gameboard.items():
            if type(piece) == King:
                kingDict[piece.Color] = position
            print(piece)
            pieceDict[piece.Color].append((piece,position))
        #white
        if self.canSeeKing(kingDict[WHITE],pieceDict[BLACK]):
            self.message = "White player is in check"
        if self.canSeeKing(kingDict[BLACK],pieceDict[WHITE]):
            self.message = "Black player is in check"


    def canSeeKing(self,kingpos,piecelist):
        #checks if any pieces in piece list (which is an array of (piece,position) tuples) can
see the king in kingpos
        for piece,position in piecelist:
            if piece.isValid(position,kingpos,piece.Color,self.gameboard):
                return True

    def parseInput(self):
        try:
```

```
            a,b = input().split()
            a = ((ord(a[0])-97), int(a[1])-1)
            b = (ord(b[0])-97, int(b[1])-1)
            print(a,b)
            return (a,b)
        except:
            print("error decoding input. please try again")
            return((-1,-1),(-1,-1))

    """def validateInput(self, *kargs):
        for arg in kargs:
            if type(arg[0]) is not type(1) or type(arg[1]) is not type(1):
                return False
        return True"""

    def printBoard(self):
        print("  1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |")
        for i in range(0,8):
            print("-"*32)
            print(chr(i+97),end="|")
            for j in range(0,8):
                item = self.gameboard.get((i,j)," ")
                print(str(item)+' |', end = " ")
            print()
        print("-"*32)



    """game class. contains the following members and methods:
    two arrays of pieces for each player
    8x8 piece array with references to these pieces
    a parse function, which turns the input from the user into a list of two tuples denoting start and end points
    a checkmateExists function which checks if either players are in checkmate
    a checkExists function which checks if either players are in check (woah, I just got that nonsequitur)
    a main loop, which takes input, runs it through the parser, asks the piece if the move is valid, and moves the piece if it is. if the move conflicts with another piece, that piece is removed. ischeck(mate) is run, and if there is a checkmate, the game prints a message as to who wins
    """


class Piece:
```

```
    def __init__(self,color,name):
        self.name = name
        self.position = None
        self.Color = color
    def isValid(self,startpos,endpos,Color,gameboard):
        if endpos in self.availableMoves(startpos[0],startpos[1],gameboard, Color =
Color):
            return True
        return False
    def __repr__(self):
        return self.name

    def __str__(self):
        return self.name

    def availableMoves(self,x,y,gameboard):
        print("ERROR: no movement for base class")

    def AdNauseum(self,x,y,gameboard, Color, intervals):
        """repeats the given interval until another piece is run into.
        if that piece is not of the same color, that square is added and
         then the list is returned"""
        answers = []
        for xint,yint in intervals:
            xtemp,ytemp = x+xint,y+yint
            while self.isInBounds(xtemp,ytemp):
                #print(str((xtemp,ytemp))+"is in bounds")

                target = gameboard.get((xtemp,ytemp),None)
                if target is None: answers.append((xtemp,ytemp))
                elif target.Color != Color:
                    answers.append((xtemp,ytemp))
                    break
                else:
                    break

                xtemp,ytemp = xtemp + xint,ytemp + yint
        return answers

    def isInBounds(self,x,y):
        "checks if a position is on the board"
        if x >= 0 and x < 8 and y >= 0 and y < 8:
            return True
        return False
```

```
    def noConflict(self,gameboard,initialColor,x,y):
        "checks if a single position poses no conflict to the rules of chess"
        if self.isInBounds(x,y) and (((x,y) not in gameboard) or gameboard[(x,y)].Color !=
initialColor) : return True
        return False


chessCardinals = [(1,0),(0,1),(-1,0),(0,-1)]
chessDiagonals = [(1,1),(-1,1),(1,-1),(-1,-1)]


def knightList(x,y,int1,int2):
    """sepcifically for the rook, permutes the values needed around a position for
noConflict tests"""
    return [(x+int1,y+int2),(x-int1,y+int2),(x+int1,y-int2),(x-int1,y-
int2),(x+int2,y+int1),(x-int2,y+int1),(x+int2,y-int1),(x-int2,y-int1)]
def kingList(x,y):
    return [(x+1,y),(x+1,y+1),(x+1,y-1),(x,y+1),(x,y-1),(x-1,y),(x-1,y+1),(x-1,y-1)]




class Knight(Piece):
    def availableMoves(self,x,y,gameboard, Color = None):
        if Color is None : Color = self.Color
        return [(xx,yy) for xx,yy in knightList(x,y,2,1) if self.noConflict(gameboard,
Color, xx, yy)]

class Rook(Piece):
    def availableMoves(self,x,y,gameboard ,Color = None):
        if Color is None : Color = self.Color
        return self.AdNauseum(x, y, gameboard, Color, chessCardinals)

class Bishop(Piece):
    def availableMoves(self,x,y,gameboard, Color = None):
        if Color is None : Color = self.Color
        return self.AdNauseum(x, y, gameboard, Color, chessDiagonals)

class Queen(Piece):
    def availableMoves(self,x,y,gameboard, Color = None):
        if Color is None : Color = self.Color
        return self.AdNauseum(x, y, gameboard, Color, chessCardinals+chessDiagonals)
```

```
class King(Piece):
    def availableMoves(self,x,y,gameboard, Color = None):
        if Color is None : Color = self.Color
        return [(xx,yy) for xx,yy in kingList(x,y) if self.noConflict(gameboard, Color, xx,
yy)]

class Pawn(Piece):
    def __init__(self,color,name,direction):
        self.name = name
        self.Color = color
        #of course, the smallest piece is the hardest to code. direction should be either 1 or
-1, should be -1 if the pawn is traveling "backwards"
        self.direction = direction
    def availableMoves(self,x,y,gameboard, Color = None):
        if Color is None : Color = self.Color
        answers = []
        if (x+1,y+self.direction) in gameboard and self.noConflict(gameboard, Color, x+1,
y+self.direction) : answers.append((x+1,y+self.direction))
        if (x-1,y+self.direction) in gameboard and self.noConflict(gameboard, Color, x-1,
y+self.direction) : answers.append((x-1,y+self.direction))
        if (x,y+self.direction) not in gameboard and Color == self.Color :
answers.append((x,y+self.direction))# the condition after the and is to make sure the
non-capturing movement (the only fucking one in the game) is not used in the
calculation of checkmate
        return answers


uniDict = {WHITE : {Pawn : "♙", Rook : "♖", Knight : "♘", Bishop : "♗", King :
"♔", Queen : "♕" }, BLACK : {Pawn : "♟", Rook : "♜", Knight : "♞", Bishop :
"♝", King : "♚", Queen : "♛" }}









Game()
```

TEST CASE 1

When the chess pieces are just moving from their current position to next position according to the rules of the piece.

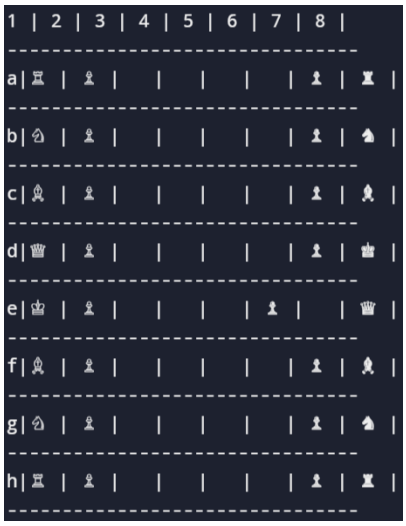

Figure 1. test case 1.1

The white piece moves and now its black’s turn.

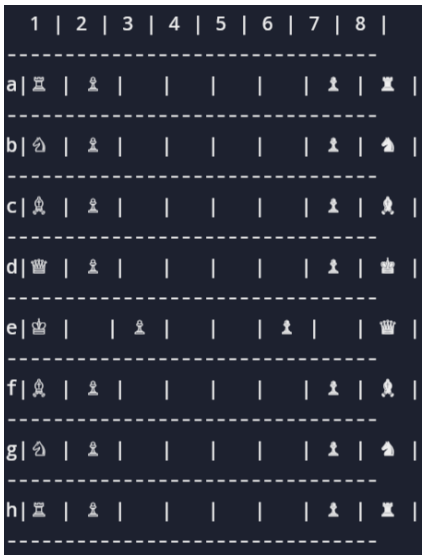

Figure 2. test case 1.2

Keep in mind that if u move into an area in which it can’t go, the program notifies the error.

Figure 3. test case 1.3

As you can see that it shows it is a invalid move and notifies it to the user. All the moves will be continued if there is no error and go smoothly. If any move is given wrong then the program notifies the user and asks for the correct moves.

Figure 4. test case 1.4

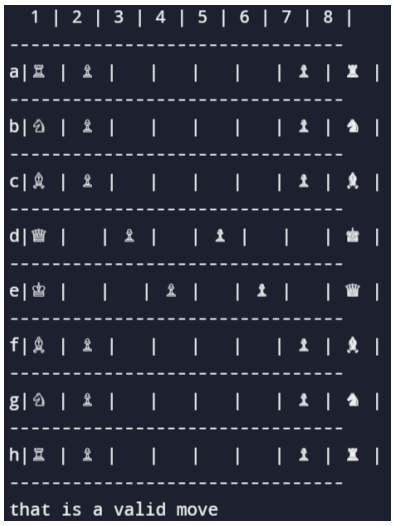

Figure 5. test case 1.5

TEST CASE 2:

When there is move which will affect the others players piece then it just takes the position of the other player position.

As you see in the below figure 2.1
The white soldier attacks the black soldier and takes the position of the black soldier.

Just like the above scenario the attacker takes the position of the defender. In any scenario in chess but when the pieces is moved to the position which it cannot travel then the program notifies the user and asks for the correct move.

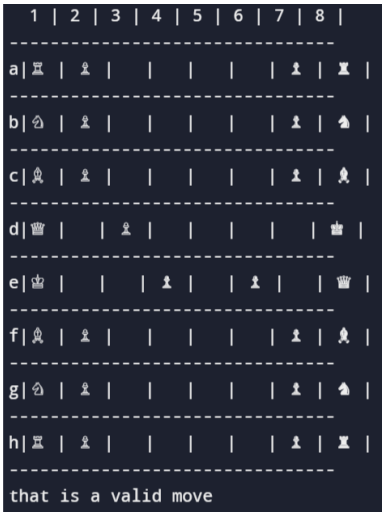

Figure 6. test case 2.1

Figure 7. test case 2.2

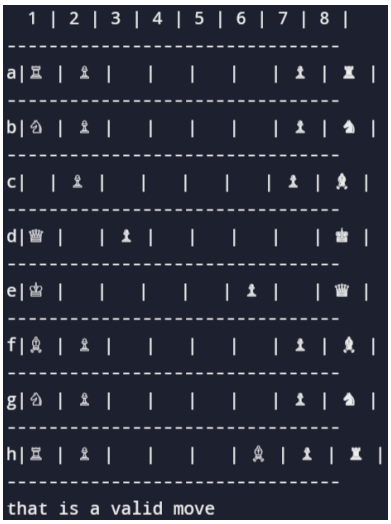

Figure 8. test case 2.3

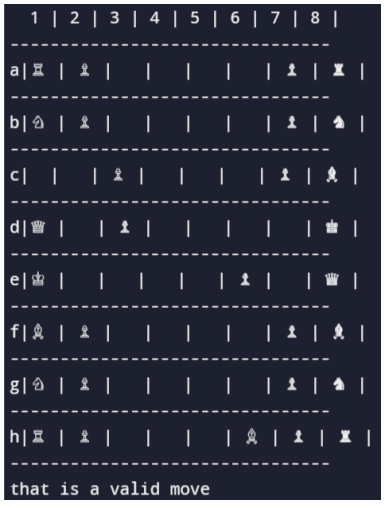

Figure 9. test case 2.4

Figure 10. test case 2.5

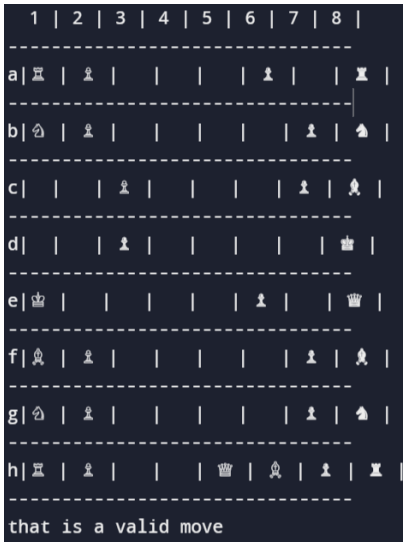

Figure 11. test case 2.6

Figure 12. test case 2.7

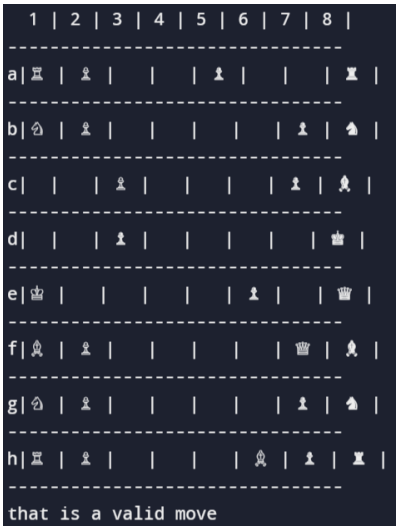

Figure 13. test case 2.8

Figure 14. test case 2.9

TEST CASE 3:

When the king checked by the other player then the program lets the user to know that their king is checked by the other player and he can do anything in his power to make his king not checked, if he fails to do so then the other player wins.

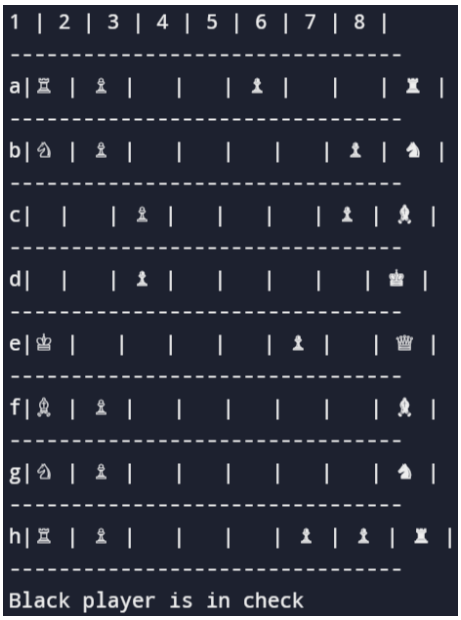

Figure 15. test case 3.1

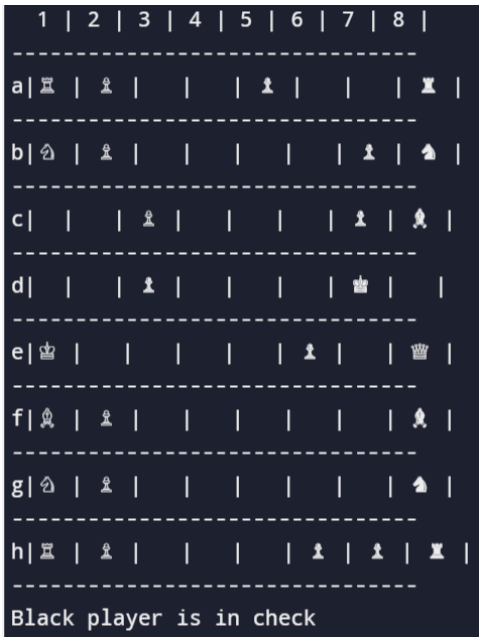

Figure 16. test case 3.2

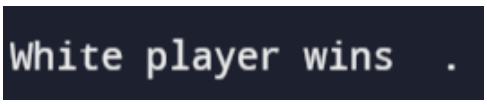

Figure 17. test case 3.3

All the moves are given in the below table:-

WHITE PLAYER'S MOVE	BLACK PLAYER'S MOVE
e7 e6	e2 e3
d7 d6	d2 d3
d6 d5	e3 e4
d5 e4	c1 h6
e4 d3	c2 d3
a7 a6	d1 h5
a6 a5	h5 f7
g7 h6	f7 e8
d8 d7	d4 e6
a6 a5	Check mate

Table no 1. White and black players moves

CHAPTER 5
CONCLUSION

In this project I modified the Beowulf chess engine to adapt to a player's skill level. After each move, the players are assigned points according to how good the move was. A total score is given to each player after the game. When the game starts both players have current position of 0. After each move, their current position changes. If the current position is positive it means that they are in the winning position and if it negatives it mean that they are in loosing position. We use this variable to create the adaptive engine. If the current position is greater than 1.3 for 3 consecutive moves the adaptive engine changes the skill level. There were 3 test cases used to test the adaptation. In the first test case, the computer played with another computer player and the weaker player adapted to the stronger player's level during the game. In the second test case, the computer played with itself and the stronger player adapted to the weaker player's level during the game. In the third test case, the computer played with a human player and adapted to the human player's skill level.

Future work for this project will include using the graphical interface (Xboardor Winboard) for the adaptive chess engine, and creating a database to save each player's moves, current positions, and score

REFERENCES

1- “Iterative deepening depth-first search” Retrieved from: http://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search [May 5, 2015].

2- “Board representation (chess)" Retrieved from: http://en.wikipedia.org/wiki/Board_representation_(chess) [December 19, 2014].

3- "a-history-of-computer-chess" Retrieved from: http://hightechhistory.com/2011/04/21/ a-history-of-computer-chess-from-the-mechanical-turk-to-Cdeep-blue/ [June 15, 2013].

4- "Project web page for the Beowulf computer chess engine" Retrieved from: http://www.frayn.net/beowulf /[June 15, 2013].

5- "Engine ratings" Retrieved from: http://computerchess.org.uk/ccrl/4040/rating_list_all.html, [May 16, 2015].

6- "Game tree" Retrieved from: http://en.wikipedia.org/wiki/Game_tre e

[June 15, 2013].

7- "Minimax" Retrieved from:
https://en.wikipedia.org/wiki/Minima
x
[June 15, 2013].

8- "Alpha–beta pruning" Retrieved from:
http://en.wikipedia.org/wiki/Alpha-beta_pruning
[June 15, 2013].

9- "Negascout" Retrieved from:
http://en.wikipedia.org/wiki/Negascout, 16 September 2013
[June 15, 2013].

10- "Negamax" Retrieved from:
http://chessprogramming.wikispaces.com/Negamax, June 1, 2013
[June 15, 2013].

11- Nil J. Nillson, "Artificial Intelligence: A New Synthesis, Morgan Kaufmann Publishers", June 1, 2013 [June 15, 2013].

12- “Position Scores and Evaluation” Retrieved from:
http://www.chess.com/forum/view/game-analysis/position-scores-and-evaluation

13- "Beowulf Computer Chess Engine" Retrieved from:

http://www.frayn.net/beowulf/

[Dec 14, 2005].

14- "Chess Engine" Retrieved from:

http://en.wikipedia.org/wiki/Chess_engine [May 4, 2015].

15- "Graphical User Interface" Retrieved from:

http://chessprogramming.wikispaces.com/GUI[2015].