California State University, San Bernardino
CSUSB ScholarWorks

Electronic Theses, Projects, and Dissertations | Office of Graduate Studies

9-2015

THE DESIGN AND IMPLEMENTATION OF AN ADAPTIVE CHESS GAME

Mehdi Peiravi
mehdi peiravi, mpeiraviusa@gmail.com

Follow this and additional works at: https://scholarworks.lib.csusb.edu/etd

 Part of the Robotics Commons

Recommended Citation

Peiravi, Mehdi, "THE DESIGN AND IMPLEMENTATION OF AN ADAPTIVE CHESS GAME" (2015). *Electronic Theses, Projects, and Dissertations*. 228.
https://scholarworks.lib.csusb.edu/etd/228

This Project is brought to you for free and open access by the Office of Graduate Studies at CSUSB ScholarWorks. It has been accepted for inclusion in Electronic Theses, Projects, and Dissertations by an authorized administrator of CSUSB ScholarWorks. For more information, please contact scholarworks@csusb.edu.

THE DESIGN AND IMPLEMENTATION OF AN ADAPTIVE CHESS GAME

A Project

Presented to the

Faculty of

California State University,

San Bernardino

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

---

by

Mehdi Peiravi

September 2015

THE DESIGN AND IMPLEMENTATION OF AN ADAPTIVE CHESS GAME

A Project

Presented to the

Faculty of

California State University,

San Bernardino

---

by

Mehdi Peiravi

September 2015

Approved by:

Haiyan Qiao, Committee Chair, Computer Science

Kerstin Voigt, Committee Member

Ernesto Gomez, Committee Member

© 2015 Mehdi Peiravi

ABSTRACT

In recent years, computer games have become a common form of entertainment. Fast advancement 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 game play 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 game play 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 game play are reviewed and adaptive learning mechanisms in computer chess game play is 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.

ACKNOWLEDGEMENTS

In performing this project, I was fortunate enough to receive wonderful guidance and assistance from some distinguished professors. I would like to show specific gratitude and thanks to Dr. Voigt for granting me admission to this institution and for kindly presiding on my committee. In addition, I would like to express my sincere appreciation to Dr. Qiao for her advice and patience with me throughout this endeavor.

On a purely human note, I would like to acknowledge Ali and Hale Hejazi for their continuous support and love. Their kindness to me when I entered this country will be forever appreciated.

TABLE OF CONTENTS

ABSTRACT iii

ACKNOWLEDGEMENTS iv

LIST OF TABLES vii

LIST OF FIGURES viii

CHAPTER ONE: INTRODUCTION 1

1.1. History of Computer Chess Play 1
1.1.1. An “Elaborate Hoax” 1
1.1.2. Artificial Intelligence in Computer Chess 3
1.1.3. Sargon Computer Chess 3
1.1.4. Deep Blue 4
1.2. Components of Computer Chess Play 4
1.2.1. Chess Engine 4
1.2.2. GUIs for Chess Games 4

CHAPTER TWO: METHODOLOGY 6

2.1. Game Tree and Chess 6
2.2. Search Algorithms 7
2.2.1. Minimax Algorithm 7
2.2.2. Alpha-Beta Pruning 8
2.2.3. NegaScout 8
2.2.4. NegaMax 9
2.3. Board Representations 9
2.3.1. Piece Lists 9
2.3.2. Array Based 9
2.3.3. 0x88 Method 10

2.3.4. Bitboard 11
CHAPTER THREE: DESIGN AND IMPLEMENTATION 12
3.1. The Beowulf Chess Engine and Difficulty Level 12
3.1.1. The Beowulf Chess Engine 12
3.1.2. Game Skill Levels 14
3.1.3. Adding New Function to Evaluate Movements 14
3.1.4. Giving the Computer Player a Difficulty Level 16
3.2. Making a Computer Play against a Computer 29
3.3. Current Position 30
3.4. Creating the Adaptive Chess Engine 31
3.5. Saving the Skill Level 32
CHAPTER FOUR: TESTING AND RESULTS 34
4.1. Testing Cases 34
4.1.1. Test Case 1 35
4.1.2. Test Case 2 47
4.1.3. Test Case 3 70
4.2. Results Analysis 89
CHAPTER FIVE: CONCLUSION 91
REFERENCES 92

LIST OF TABLES

Table 1. Test Case 1 47

Table 2. Test Case 2 69

Table 3. Test Case 3 88

LIST OF FIGURES

Figure 1.Game Starts Test Case 1 35
Figure 2.Screenshot 1 Test Case 1 36
Figure 3.Screenshot 2 Test Case 1 37
Figure 4.Screenshot 3 Test Case 1 38
Figure 5.Screenshot 4 Test Case 1 39
Figure 6.Screenshot 5 Test Case 1 40
Figure 7.Screenshot 6 Test Case 1 41
Figure 8.Screenshot 7 Test Case 1 42
Figure 9.Screenshot 8 Test Case 1 43
Figure 10.Screenshot 9 Test Case 1 44
Figure 11.Screenshot 10 Test Case 1 45
Figure 12.Screenshot 11 Test Case 1 46
Figure 13.Game Starts Test Case 2 48
Figure 14.Screenshot 1 Test Case 2 49
Figure 15.Screenshot 2 Test Case 2 50
Figure 16.Screenshot 3 Test Case 2 51
Figure 17.Screenshot 4 Test Case 2 52
Figure 18.Screenshot 5 Test Case 2 53
Figure 19.Screenshot 6 Test Case 2 54
Figure 20.Screenshot 7 Test Case 2 56
Figure 21.Screenshot 8 Test Case 2 57

Figure 22.Screenshot 9 Test Case 2 58
Figure 23.Screenshot 10 Test Case 2 59
Figure 24.Screenshot 11 Test Case 2 60
Figure 25.Screenshot 12 Test Case 2 61
Figure 26.Screenshot 13 Test Case 2 62
Figure 27.Screenshot 14 Test Case 2 63
Figure 28.Screenshot 15 Test Case 2 64
Figure 29.Screenshot 16 Test Case 2 65
Figure 30.Screenshot 17 Test Case 2 66
Figure 31.Screenshot 18 Test Case 2 67
Figure 32.Screenshot 19 Test Case 2 68
Figure 33.Game Starts Test Case 3 70
Figure 34.Screenshot 1 Test Case 3 71
Figure 35.Screenshot 2 Test Case 3 72
Figure 36.Screenshot 3 Test Case 3 73
Figure 37.Screenshot 4 Test Case 3 74
Figure 38.Screenshot 5 Test Case 3 75
Figure 39.Screenshot 6 Test Case 3 76
Figure 40.Screenshot 7 Test Case 3 77
Figure 41.Screenshot 8 Test Case 3 78
Figure 42.Screenshot 9 Test Case 3 79
Figure 43.Screenshot 10 Test Case 3 80

Figure 44.Screenshot 11 Test Case 3 81

Figure 45.Screenshot 12 Test Case 3 82

Figure 46.Screenshot 13 test Case 3 83

Figure 47.Screenshot 14 Test Case 3 84

Figure 48.Screenshot 15 Test Case 3 85

Figure 49.Screenshot 16 Test Case 3 86

Figure 50.Screenshot 17 Test Case 3 87

Figure 51.Screenshot 18 Test Case 3 88

CHAPTER ONE

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 [3]" 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.[3] 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 move until 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 it to search four moves ahead.[3]

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 Technology that 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 it took 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 written on 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 was the 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 TWO

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 a value 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.

2.2. 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 by the 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 also think 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 be sure 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 implement some 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 off board. 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 THREE

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 of the 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 of the following files:

Main.c

This is the main file for the chess game. It includes the main structure for the 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.

Comp.c

This file contains all the functions that calculate the computer movement. Each time the computer moves, it calls the Comp function to analyze the board

position to find out the best move. This function uses the parameters in Params and calculates the best move for the current side.

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.

Rand64.c

This is a pseudo-random number generator for 64-bit machines. It is based on Isaac64.

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 value of 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
```

```
    /* Use the recursive negascout algorithm to find the score.  If we already
     * have an estimate of the true score then use this and search only in a
window
     * around it. */
   if (!USE_WINDOW || depth==2)
     score = Search(B,-
CMSCORE,CMSCORE,depth*ONEPLY,0,inchk,InitFifty,0,NO_MOVE);

   else {
     RootAlpha = PreviousScore - WINDOW;
     RootBeta = PreviousScore + WINDOW;
     LastPlyNodecount = Nodecount;
     score =
Search(B,RootAlpha,RootBeta,depth*ONEPLY,0,inchk,InitFifty,0,NO_MOVE);
     /* If this aspiration search fails low or high then search it properly with
      * safer bounds instead.  Also increase the window size.  */
     if (score<=RootAlpha || score >= RootBeta) {
      if (!AbortFlag  && !IsCM(score)) {
        if (score >= RootBeta && Post) PrintThinking(score,B);
        if (score <= RootAlpha) {RootAlpha = -CMSCORE; RootBeta = score+1;}
        if (score >= RootBeta) {RootBeta = CMSCORE; RootAlpha = score-1;}
        score =
Search(B,RootAlpha,RootBeta,depth*ONEPLY,0,inchk,InitFifty,0,NO_MOVE);
        /* If the second search also fails with a cutoff (strangely) then we must
         * do a full re-search with infinite bounds.  The expectation is that this
         * happens extremely rarely.  Of course, it shouldn't happen at all, ideally.
         * The fact that it sometimes does is due to a phenomenon called "search
         * instability", and it's a complete pain!  At least we'll have a full hash
```

```
         * 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've
thought
   * 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 commands
  if (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 is the same as Comp function but takes a “j” parameter. This parameter is assigned to depth.

3.2. 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, we call different functions. This makes the computer move for the black player.

3.3. 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 has clear 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 continue on. 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.

3.4. 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 position for 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 them each 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 player to adapt to the black player during the game.

3.5. 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 that the 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 of the game play in order to see the adaptive process.

CHAPTER FOUR

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.

4.1.1. Test Case 1

In the first test case, the computer player (white player) plays with another computer player (back player) and adapts to the black player's skill level, which is 5. In this test case, the computer player (the white player) is used to simulate the human player. The objective of this test case is to test whether the white player (adaptive engine) adapts to the black player during the game. The initial skill level for the black player is 5. When the game starts we can choose the skill level for the white player before adaption which can be a number from 1 to 6 (1 is the easiest skill level and 6 is the strongest).

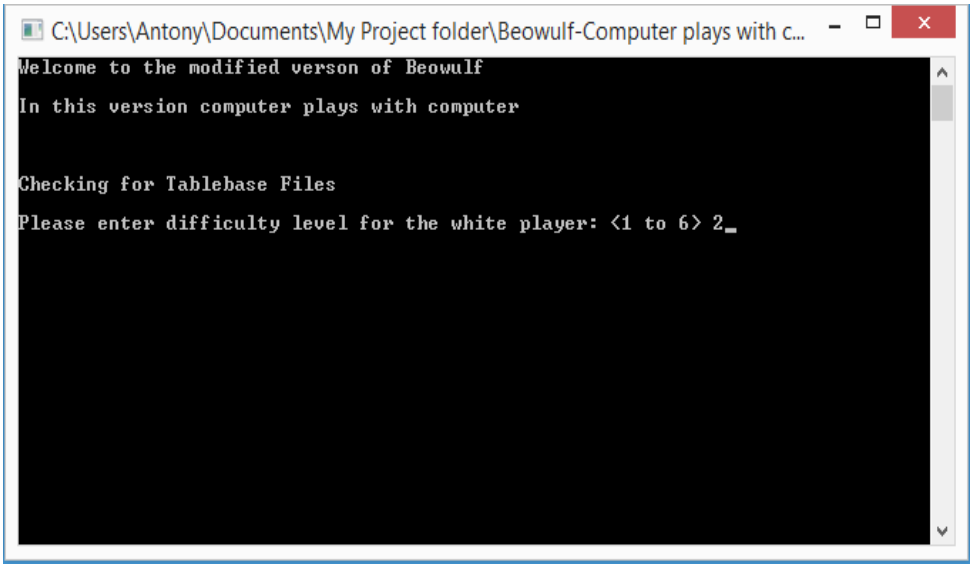

Figure 1.Game Starts Test Case 1

As shown in Figure 1, we select 2 as the skill level for the white player (adaptive engine) and the game starts. The white player makes the first move (b1c3). The current position is 0 and both players have an equal score of 39.

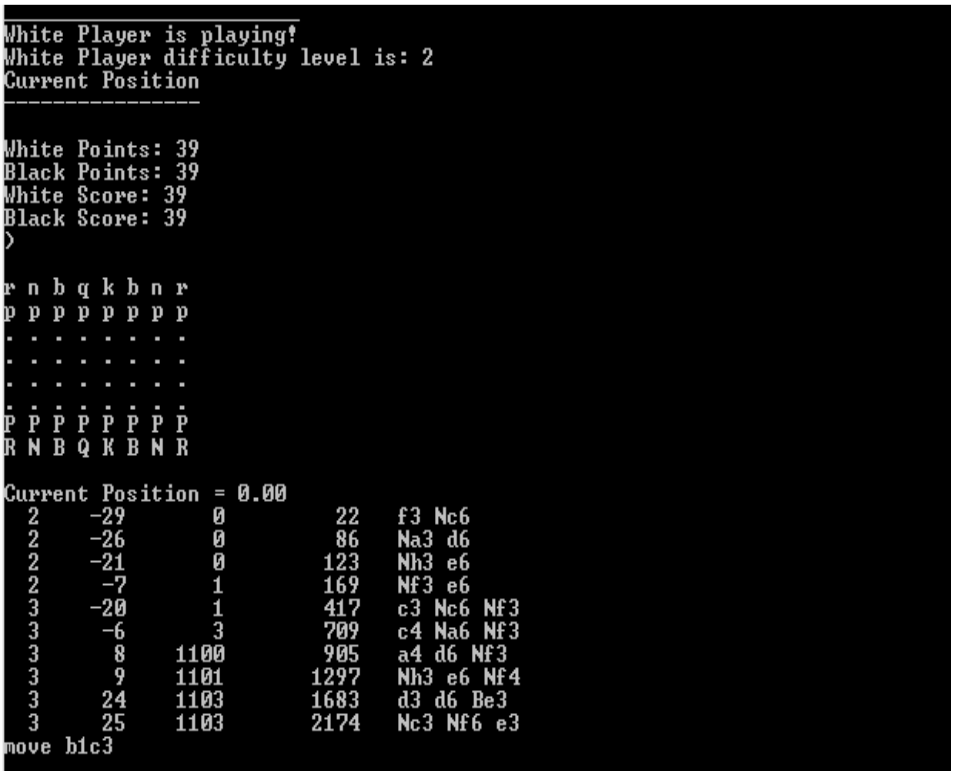

Figure 2.Screenshot 1 Test Case 1

As shown in Figure 3, the black player makes a move (c7c6). The Black player's difficulty level is 5. Then white player moves (c3b5). Both players have the same score at this time, and the current position is near to 0 (0.38).

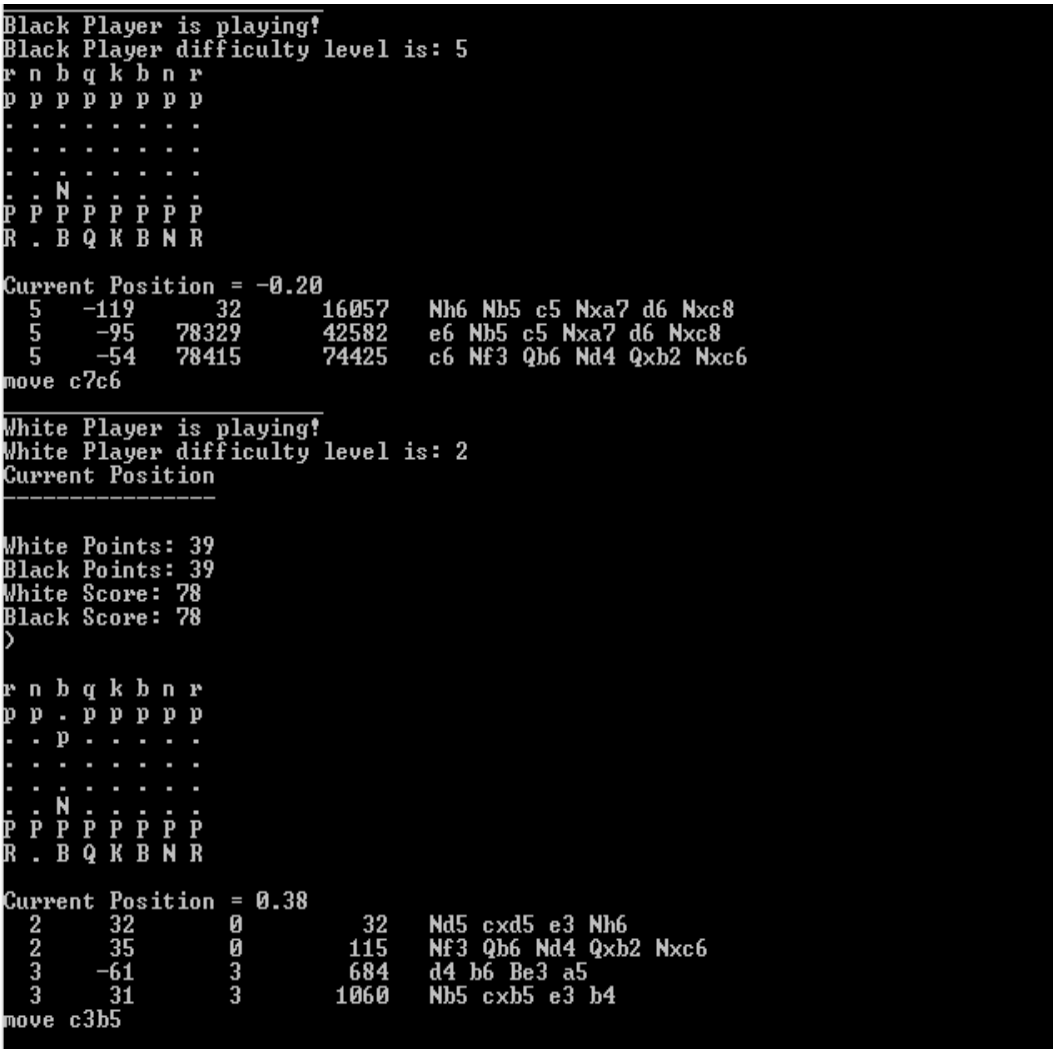

Figure 3.Screenshot 2 Test Case 1

After several moves, the current position for the black player becomes 0.5 and for the white player (adaptive engine) it becomes -2.09. This shows that the black player is in a better position than the white player. The black player's score is 346 and the white player's score is 318. This is shown in Figure 4.

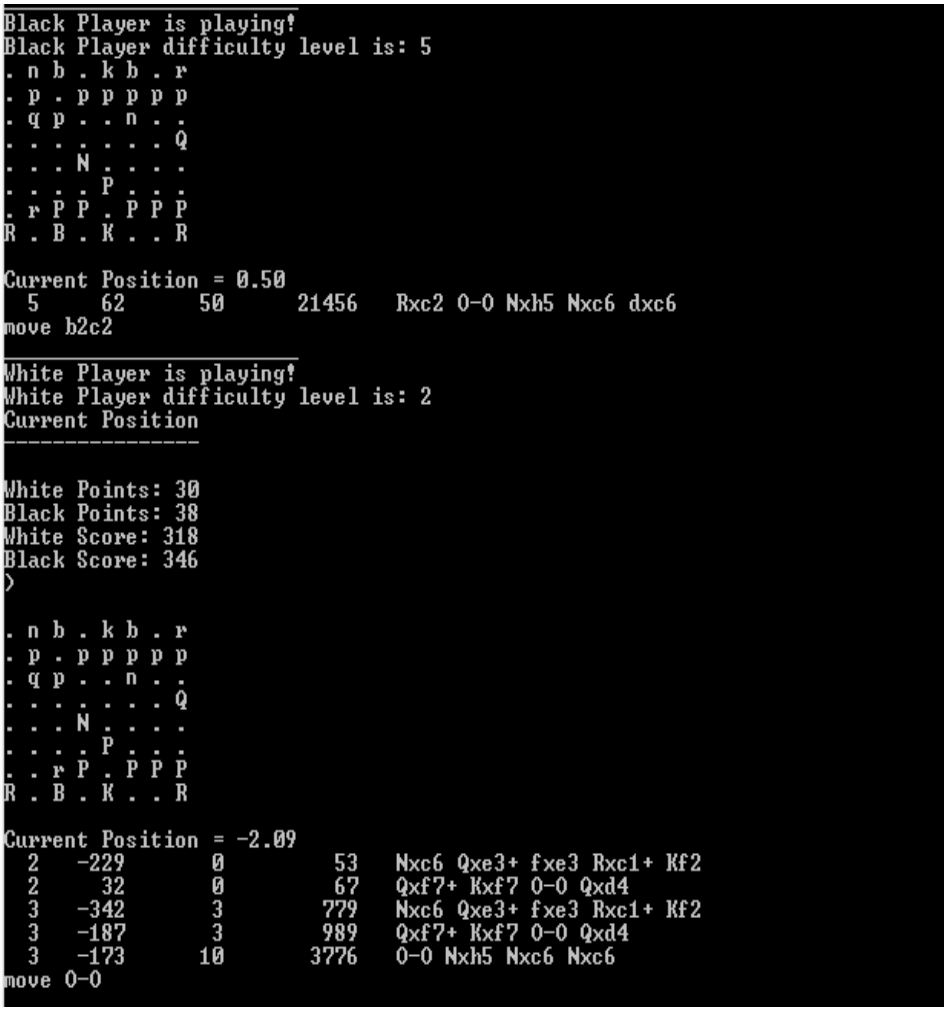

Figure 4.Screenshot 3 Test Case 1

As shown in Figure 5, the black player makes another move (f6h5) and its current position becomes 1.93. The current position for the white player is -1.55. This means that the black player is in a better position than the white player and he is more likely to win the game.

Figure 5.Screenshot 4 Test Case 1

As shown in Figure 6, the black player makes another move (b8a6) and the white player (adaptive engine) also makes a move (e3d4). The black player's score is 460 and the white player's score is 370. The current position for the black player is 1.49 and for the white player it is -1.39. We can see that the black player has a clear advantage over the white player.

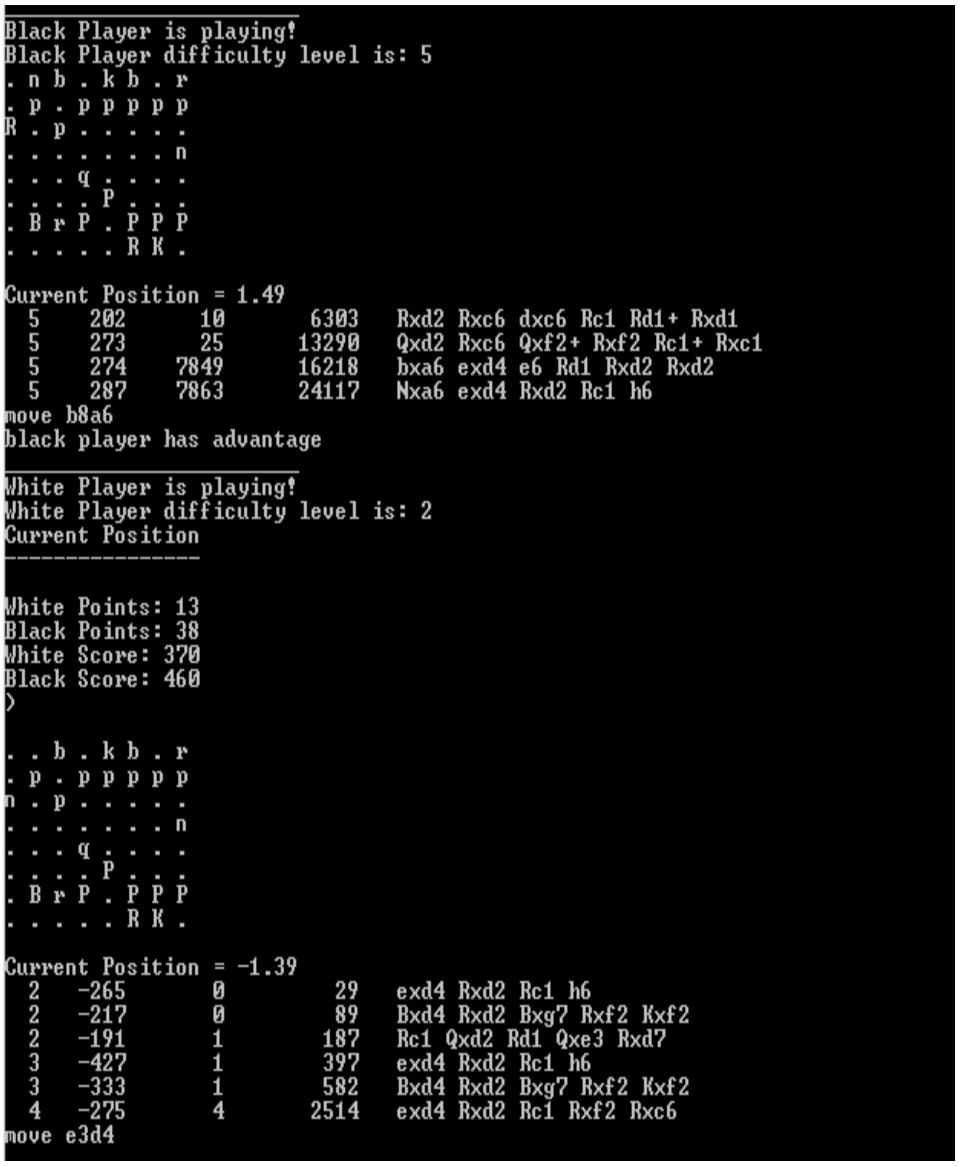

Figure 6.Screenshot 5 Test Case 1

Since the black player is winning the game, the skill level for the white player (adaptive engine) increases to 3 to adapt to the black player. The black player makes a move (c2d2) and the white player makes a move (f1c1). The

black player's score is 489 and the white player's score is 382. This is shown in Figure 7.

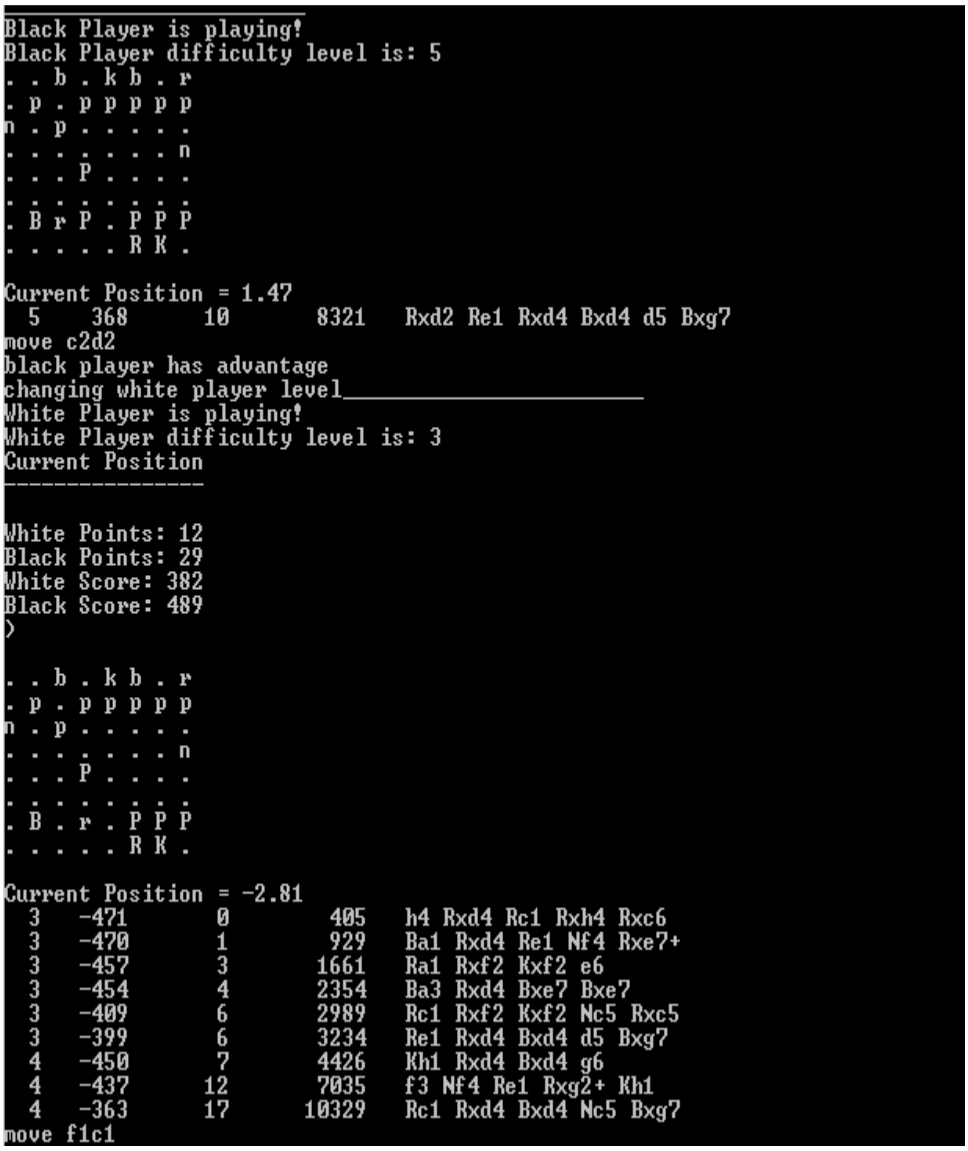

Figure 7.Screenshot 6 Test Case 1

As shown in Figure 8, after several moves, the black player still has a clear advantage. The white player's skill level changes from 3 to 4 to adapt to the

stronger player. The current position for the black player is 4.41 and for the white player it is -4.25. The black player's score is 576 and the white player's score is 416.

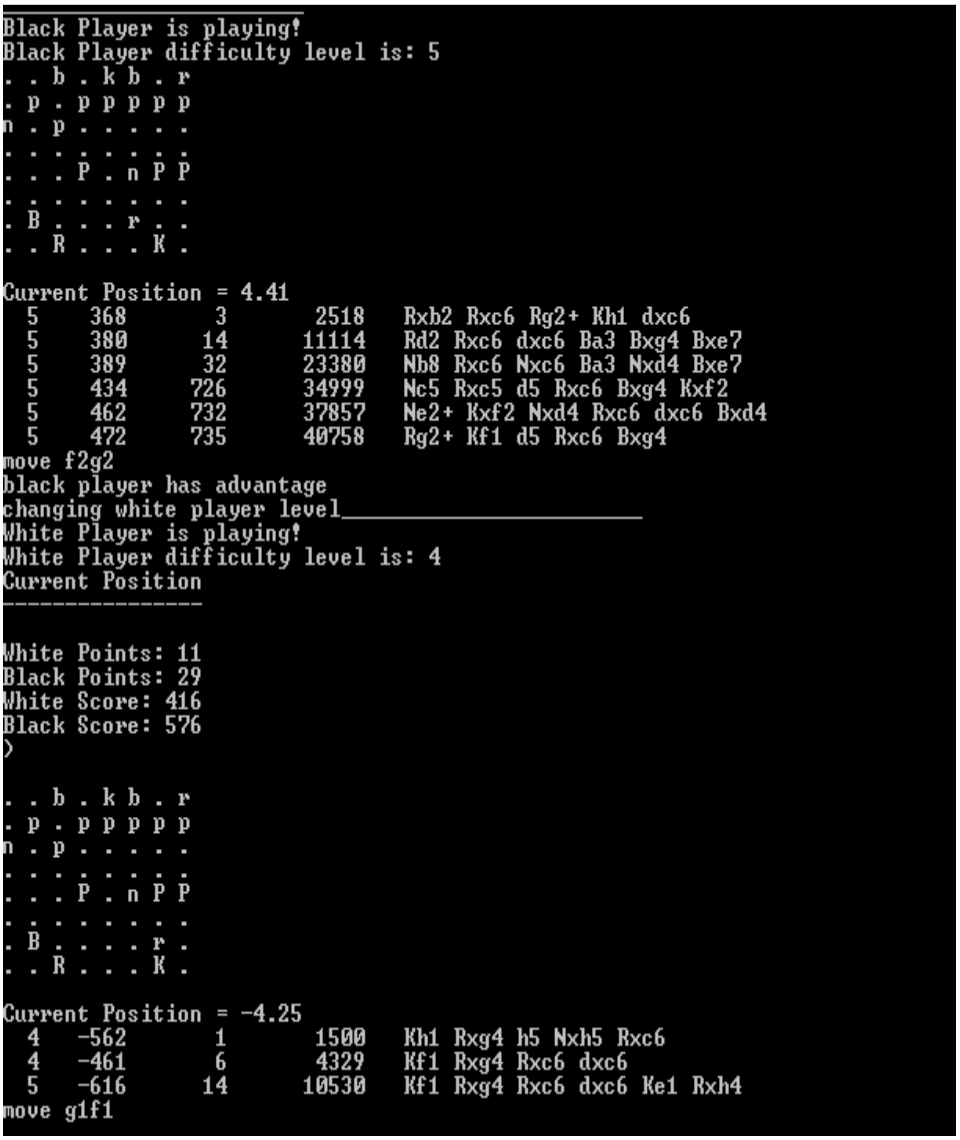

Figure 8.Screenshot 7 Test Case 1

Since the black player still has a clear advantage, the white player's skill level changes from 4 to 5. Now both players are playing at the same difficulty level. This is shown in Figure 9.

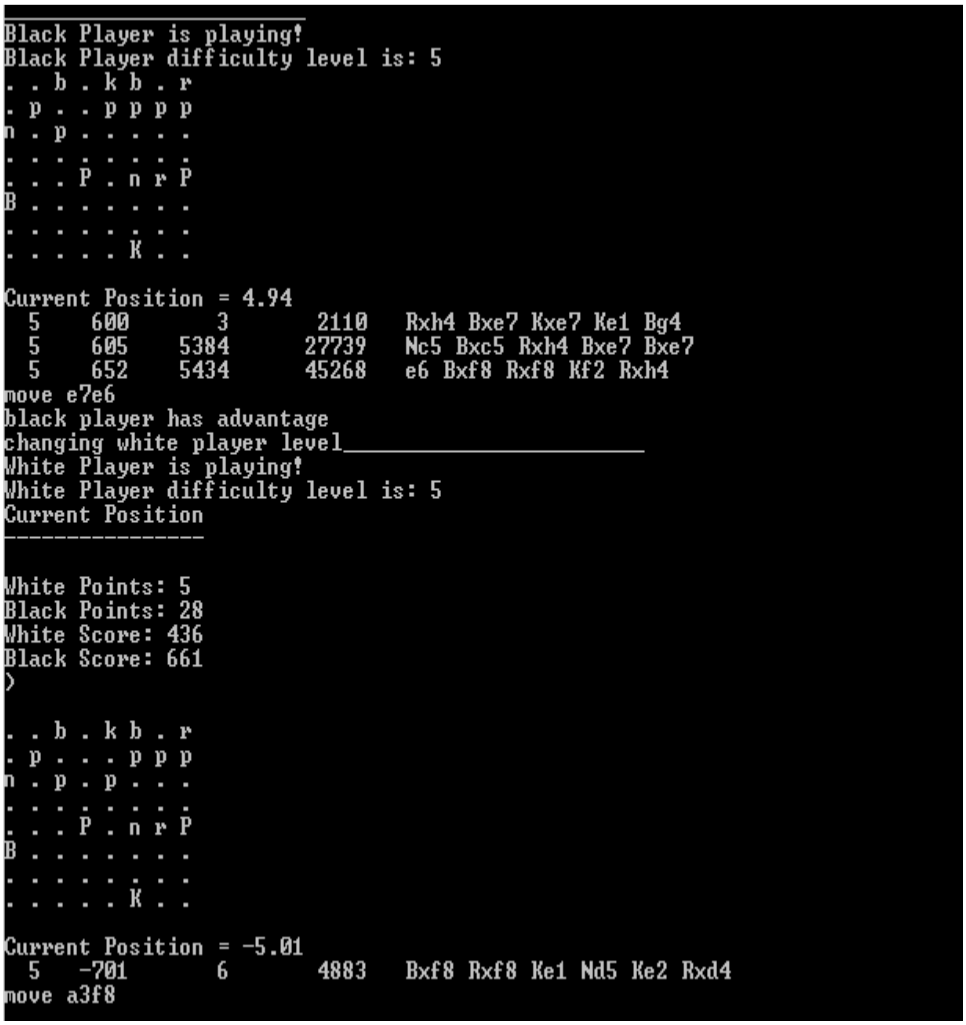

Figure 9.Screenshot 8 Test Case 1

As shown in Figure 10, both players are playing at the same skill level. The black player's current position has changed to 6.84 and the white player's

current position has changed to -108.83. This shows that the white player (adaptive engine) is losing the game.

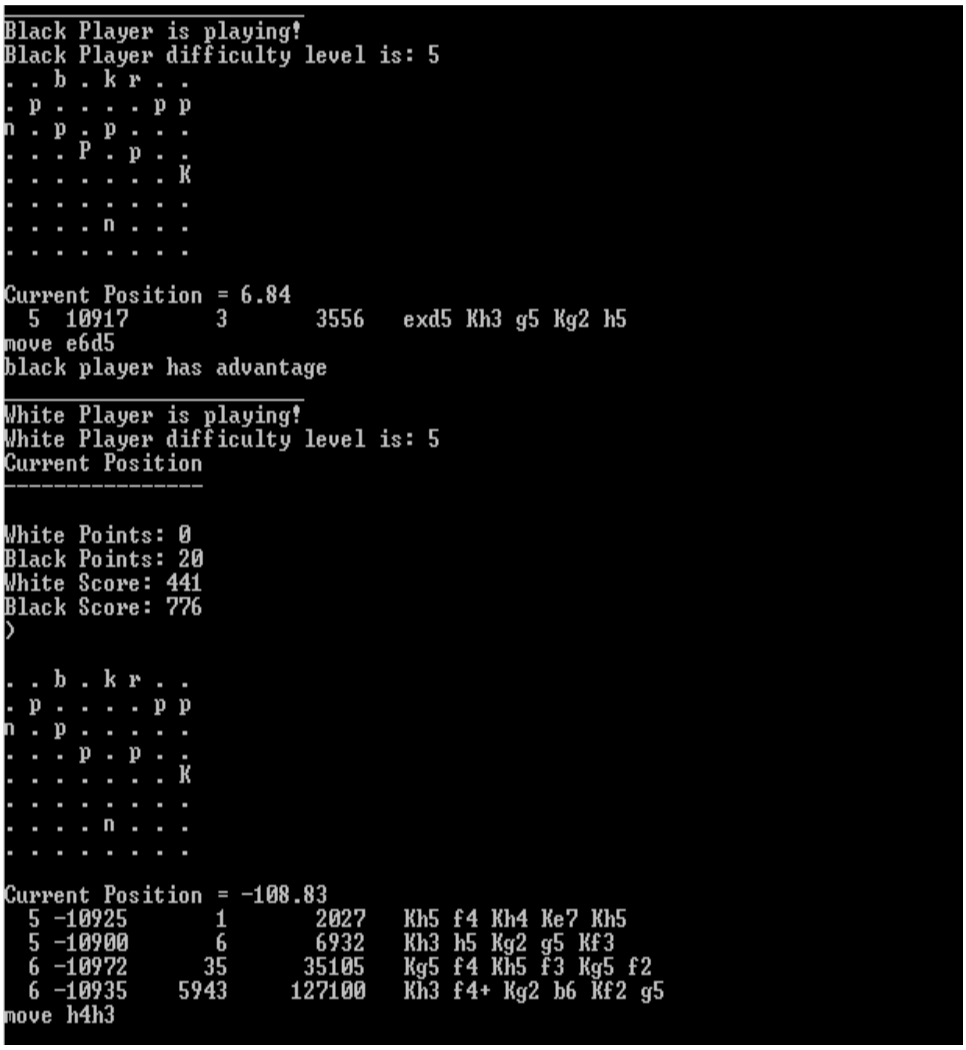

Figure 10.Screenshot 9 Test Case 1

Finally, the black player wins the game with the move e6d5. The white player's skill level after adaption is 5. The black player's score is 999 and the white player's score is 441. The game asks the user if he wants to continue playing the game or quit. This is shown in Figure 11.

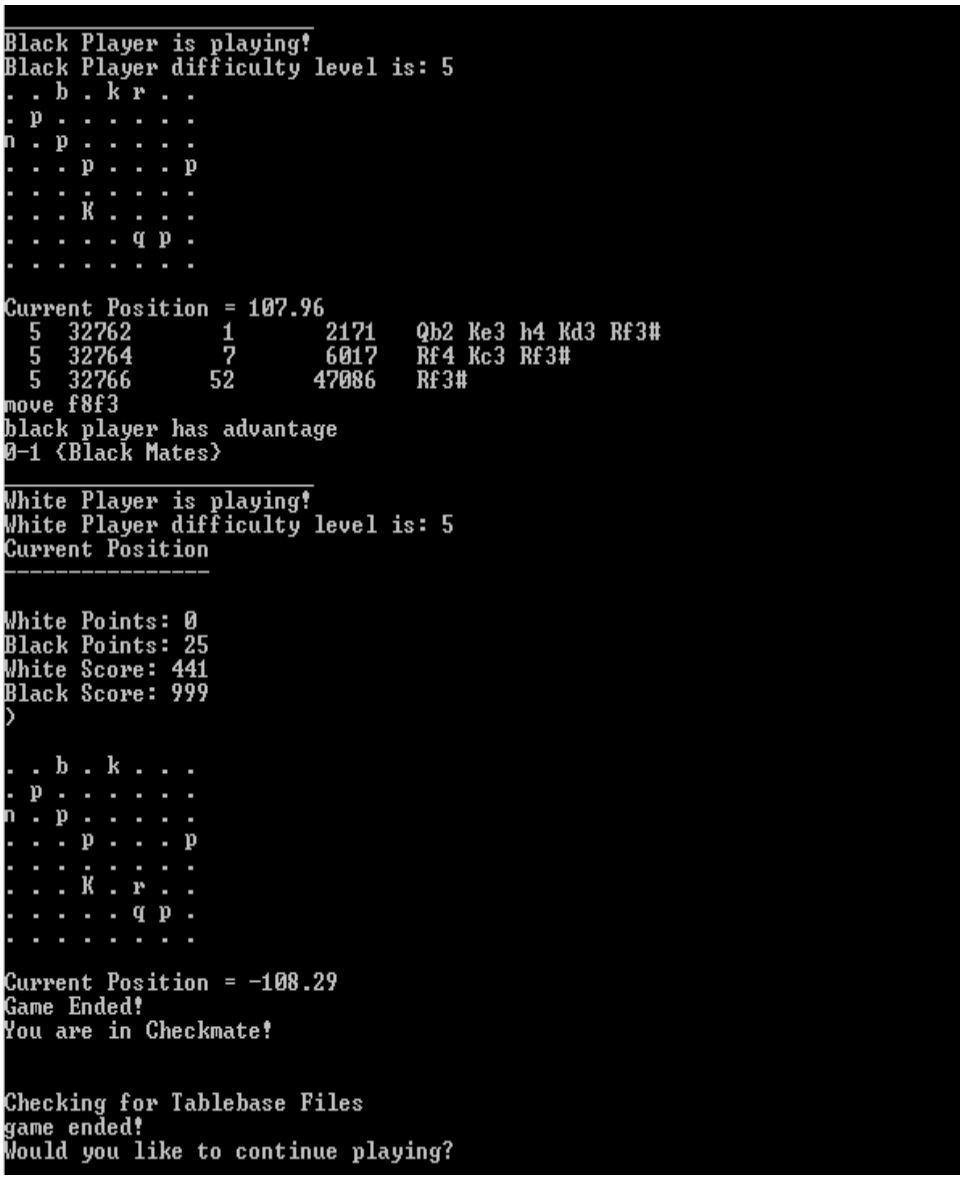

Figure 11.Screenshot 10 Test Case 1

If the user chooses to continue the game, the game starts again and the white player plays at the adapted skill level, which is 5. This is shown in Figure 12.

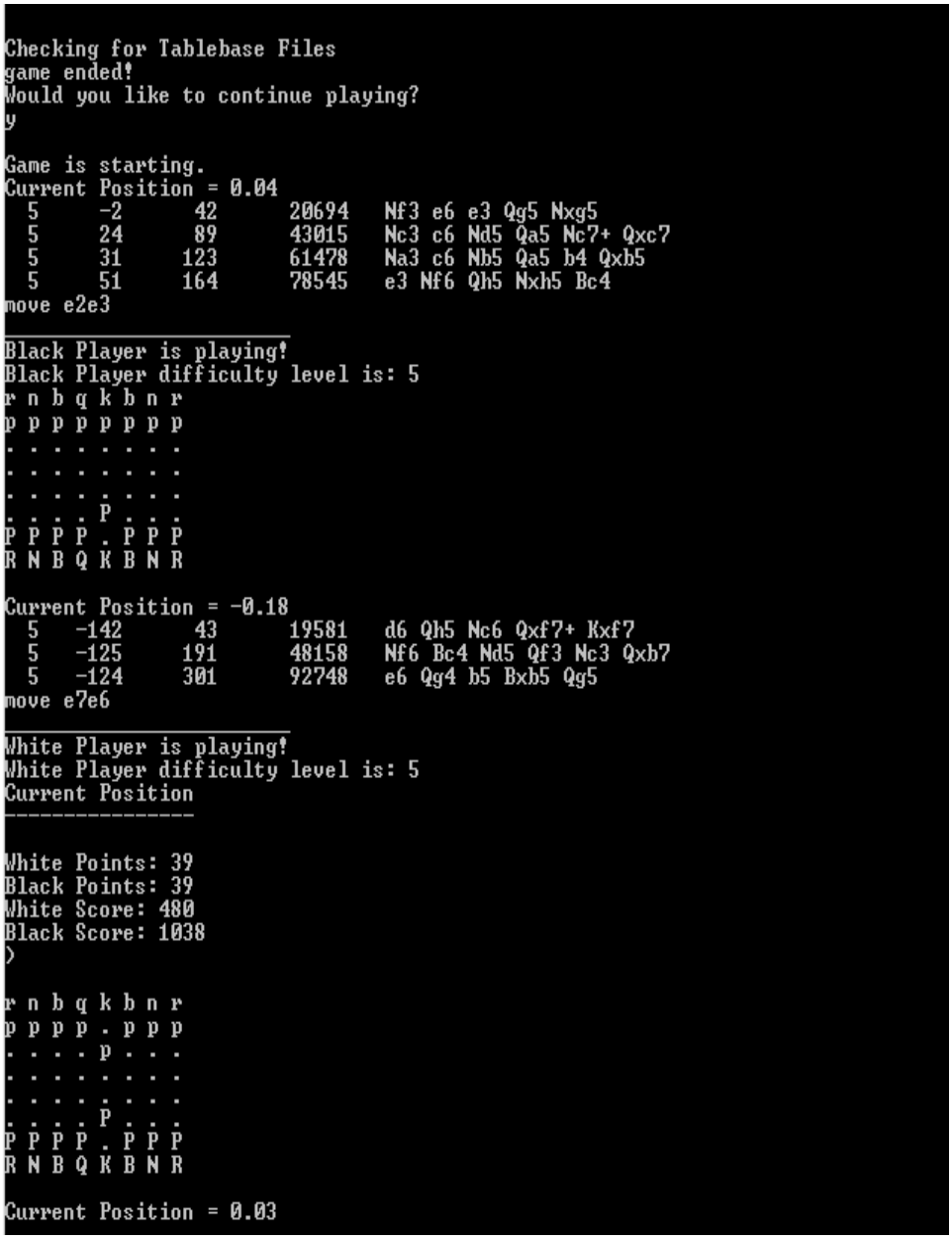

Figure 12.Screenshot 11 Test Case 1

Table 1. Test Case 1

White Player's Move	Black Player's Move	White player's difficulty level	Black Player's difficulty level	White Player's current position	Black player's current Position	White Player's Score	Black Player's Score
B1C3	C7C6	2	5	0	0	39	39
C3B5		2	5	0.38	-0.20	78	78
O-O	B2C2	2	5	-2.09	0.50	318	346
C1B2	F6H5	2	5	-1.55	1.93	339	384
E3D4	B8A6	2	5	-1.39	1.49	370	460
F1C1	C2D2	3	5	-2.81	1.47	382	489
G1F1	F2G2	4	5	-4.25	4.41	416	576
A3F8	E7E6	5	5	-5.01	4.94	436	661
H4H3	E6D5	5	5	-108.83	6.84	441	776

4.1.2. Test Case 2

In the second test case, the computer player (white player) plays with another computer player (black player) and it adapts to the black player's skill level, which is 1. In this test case, a computer player is used to simulate the human player. The objective of this test case is to test whether the white player (adaptive engine) adapts to the black player during the game. The following screenshots show how this adaptation happens. When the game starts, the

computer prompts the user to enter the skill level for the white player. This skill level can be a number from 1 to 6.

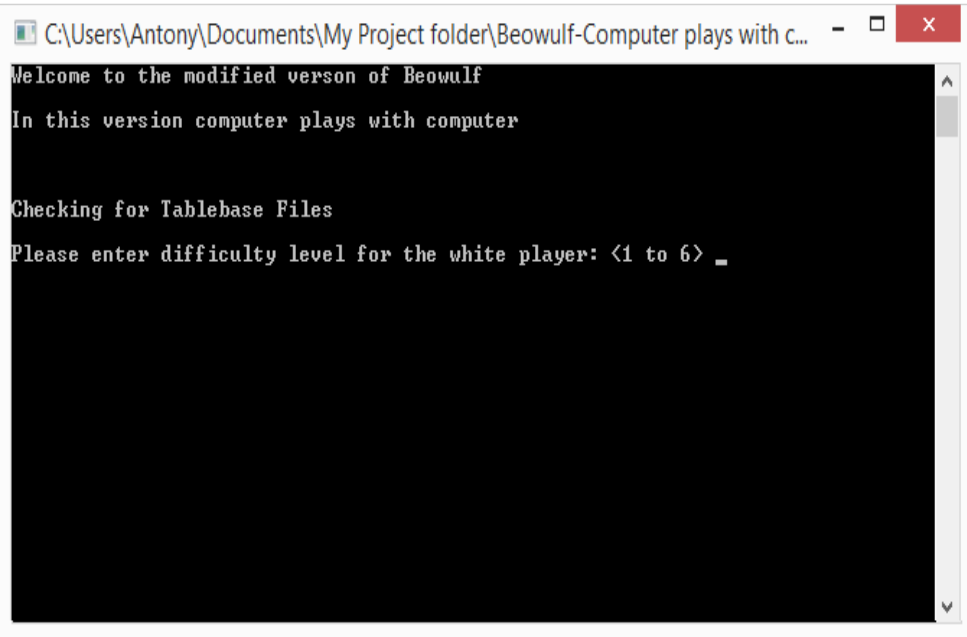

Figure 13.Game Starts Test Case 2

As shown in Figure 14, the user enters 5 for the white player's difficulty level. The game starts and the black player's difficulty level is 1. The white player (adaptive engine) makes the move c2c3 and the black player makes the move b8c6. The current position for the white player is 0 and for the black player it is 0.11. Both players have an equal score of 39.

Figure 14.Screenshot 1 Test Case 2

The white player (adaptive engine) makes the second move d1b3 and the black player makes the move f7f6. Both players have an equal score of 78. The current position for the white player is -0.34 and for the black player it is 0.51.

This is shown in Figure 15.

Figure 15.Screenshot 2 Test Case 2

As shown in Figure 16, the white player (adaptive engine) makes the third move b3b7 and the black player makes the move c8b7. Both players' still have the same score of 117. The current position for the white player is -0.47 and for the black player it is -0.63.

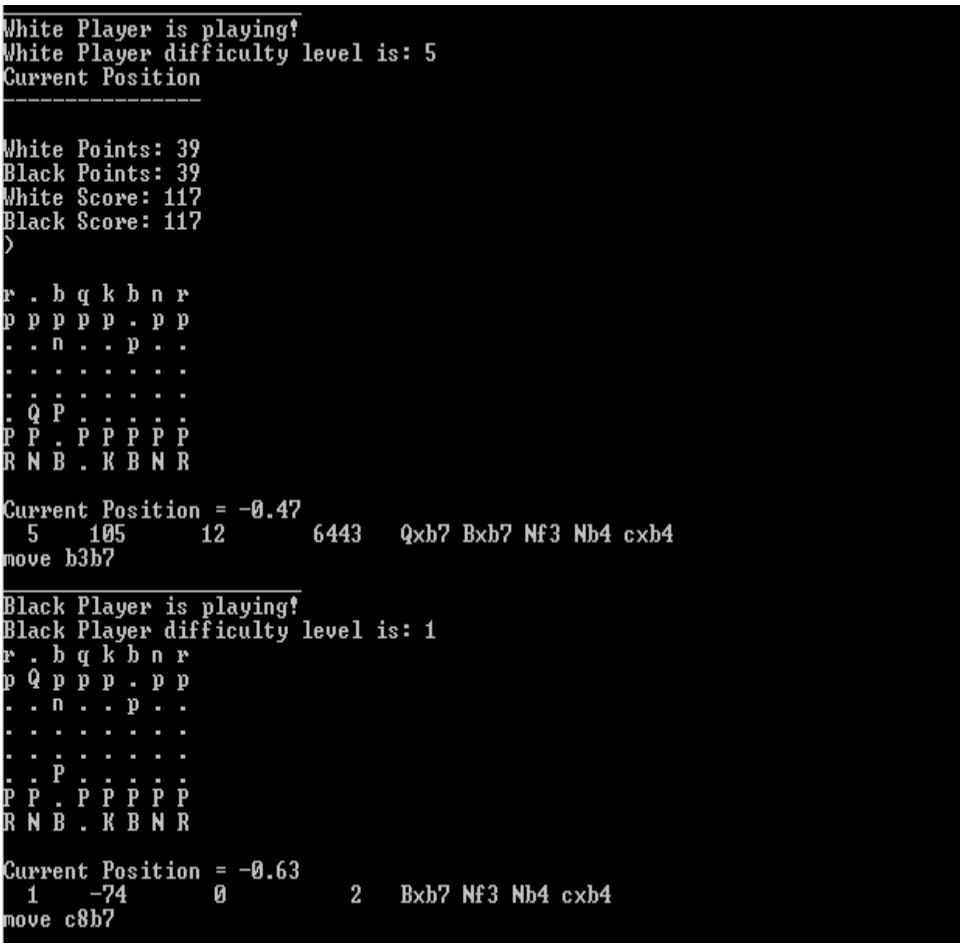

Figure 16.Screenshot 3 Test Case 2

As shown in Figure 17, the white player (adaptive engine) makes the move g1f3 and the black player makes the move e7e6. The current position for the white player increases to 0.80 and for the black player it decreases to -0.98. This shows that the white player's position keeps getting better in the game. The white player's score is 147 and the black player's score is 155. The white player

still has a skill level of 5 and the black player's skill level is 1.

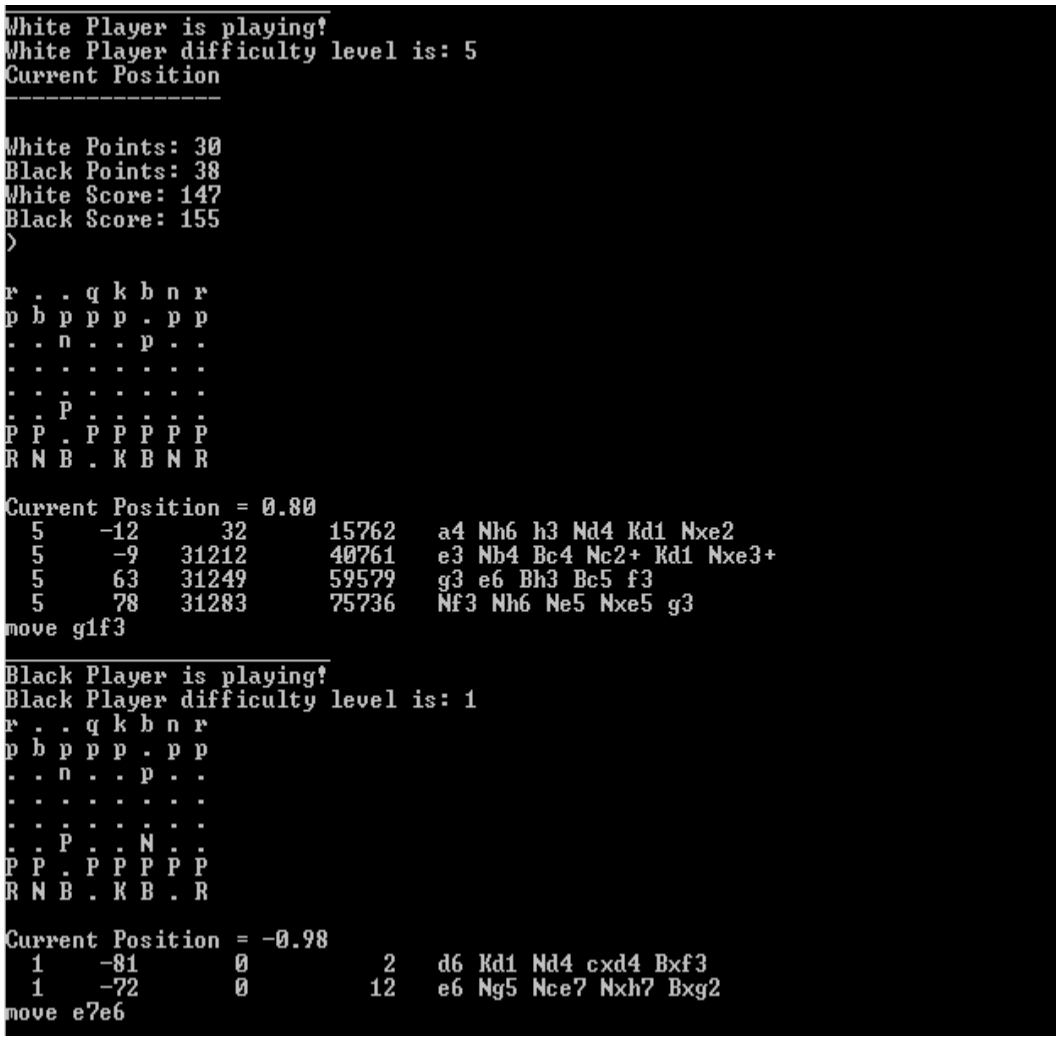

Figure 17.Screenshot 4 Test Case 2

The white player (adaptive engine) makes the next move f3g5 and the black player responds with the move f6g5. The current position for the white player is 1.01 and for the black player it is -0.95. Since the white player has a current position greater than 1, the white player has a clear advantage over the black player. The white player's score is 177 and the black player's score is 193.

The white player still has a skill level of 5 and the black player has a skill level of 1.This is shown in Figure 18.

Figure 18.Screenshot 5 Test Case 2

As shown in Figure 19, the white player (adaptive engine) makes the move d2d3 and the black player responds with g5g4. The current position for the white player is 0.88 and for the black player it is -1.17. The white player's score is

204 and black player's score is 231. The white player's skill level is still 5 and the black player's skill level is 1. Since the current position for the black player is decreasing, it shows that the black player is more likely to lose the game.

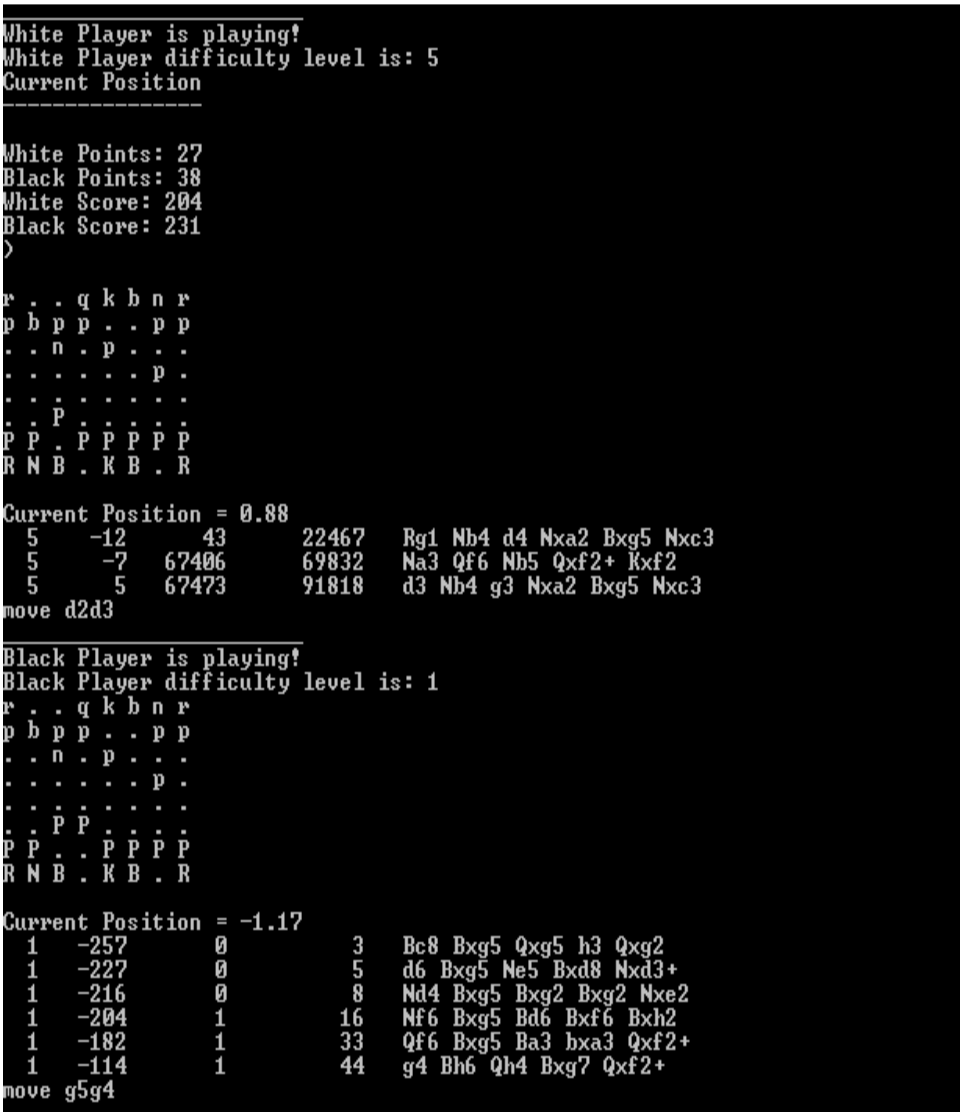

Figure 19.Screenshot 6 Test Case 2

The white player (adaptive engine) makes the move c1e3 and the black player makes the move a7a6. The current position for the white player changes to 1.16 and for the black player it changes to -1.28. The white player's score is 231 and the black player's score is 269. The skill level for the white player is still 5 and for the black player it is 1. Since the white player's current position is 1.16, he has a clear advantage over the black player. This is shown in Figure 20.

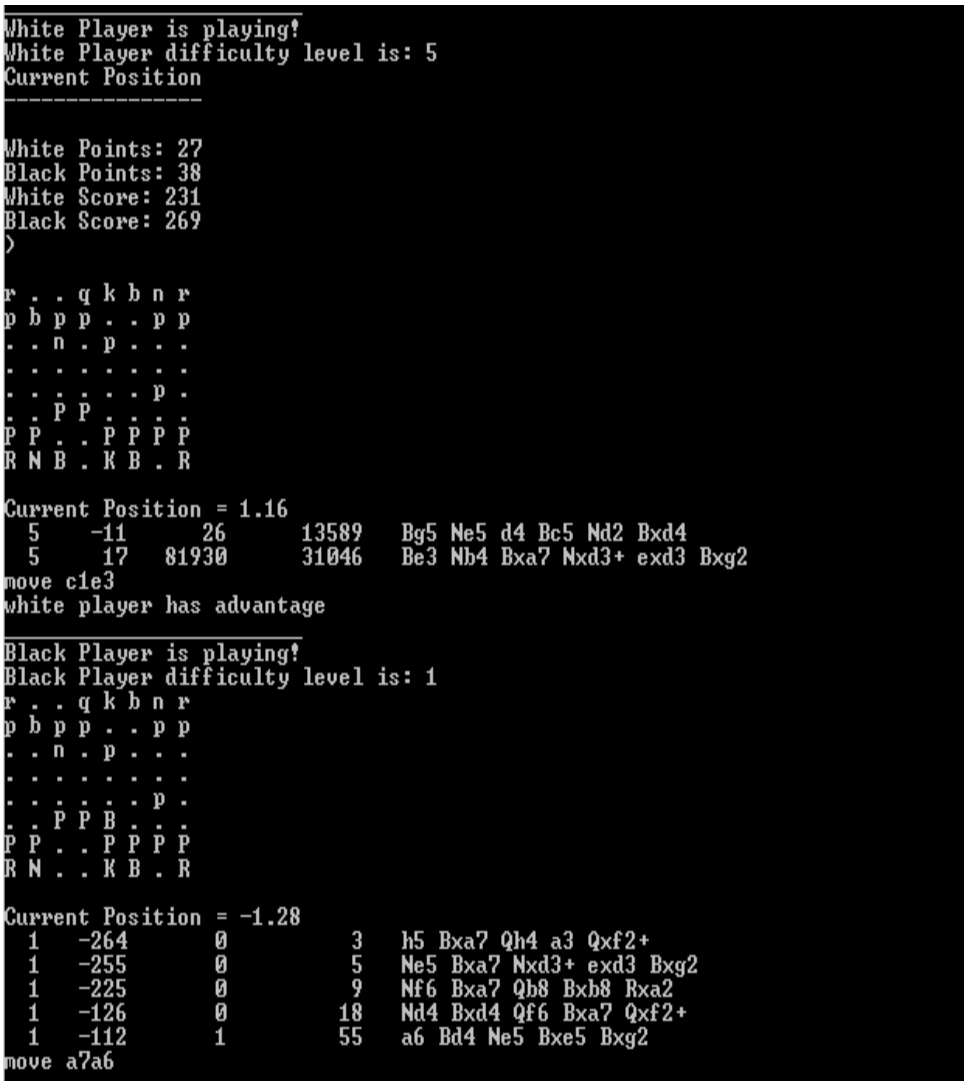

Figure 20.Screenshot 7 Test Case 2

As shown in Figure 21, the white player (adaptive engine) makes the next move b1d2 and the black player makes the move g8e7. The current position for the white player increases to 1.21 and for the black player it decreases to -1.4. The white player's skill level is still 5 and the black player's skill level is 1.

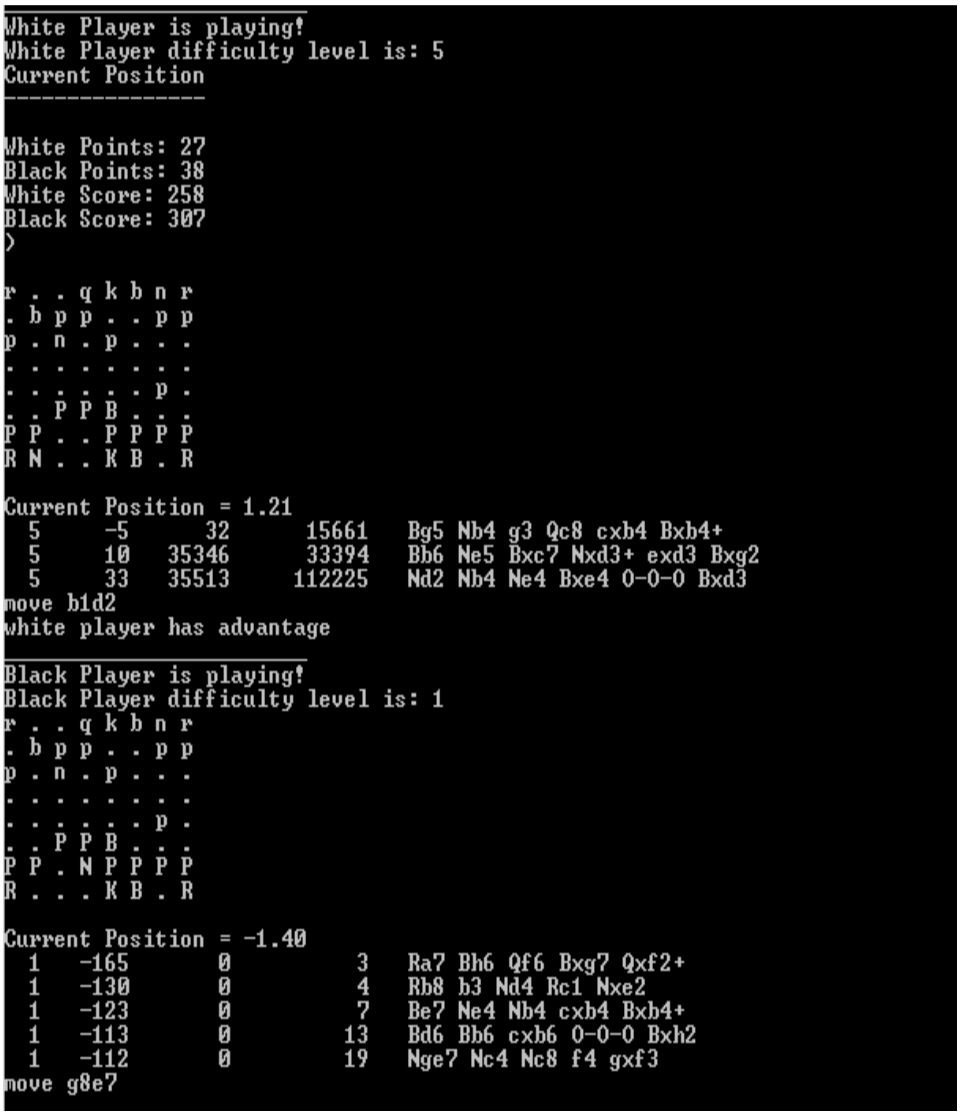

Figure 21.Screenshot 8 Test Case 2

After several moves, the white player's current position changes to 1.48 and the black player's current position changes to -1.49. This shows that the

white player is in a winning position. The white player (adaptive engine) still has a skill level of 5 and the black player's skill level is 1. This is shown in Figure 22.

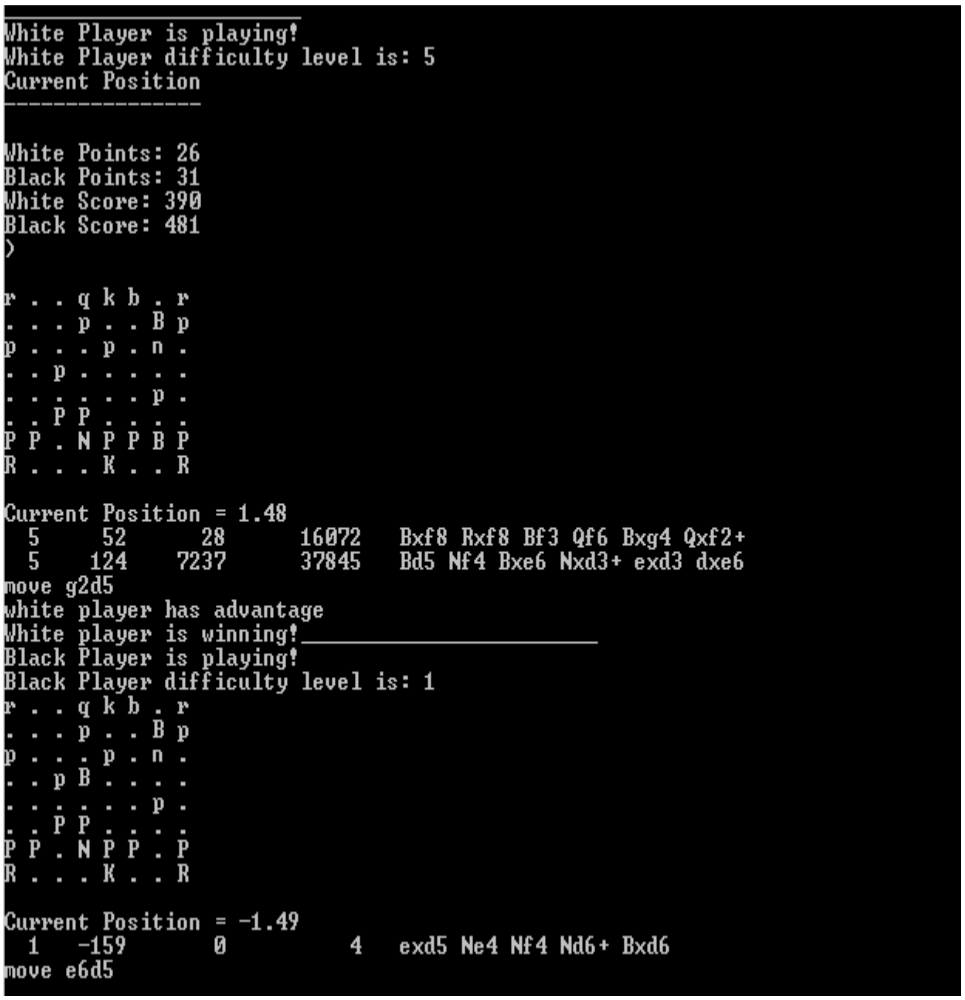

Figure 22.Screenshot 9 Test Case 2

As show in Figure 23, the white player makes the move h1g1 and the black player makes the move d7d6. The current position for the white player is

1.3 and for the black player it is -1.41.

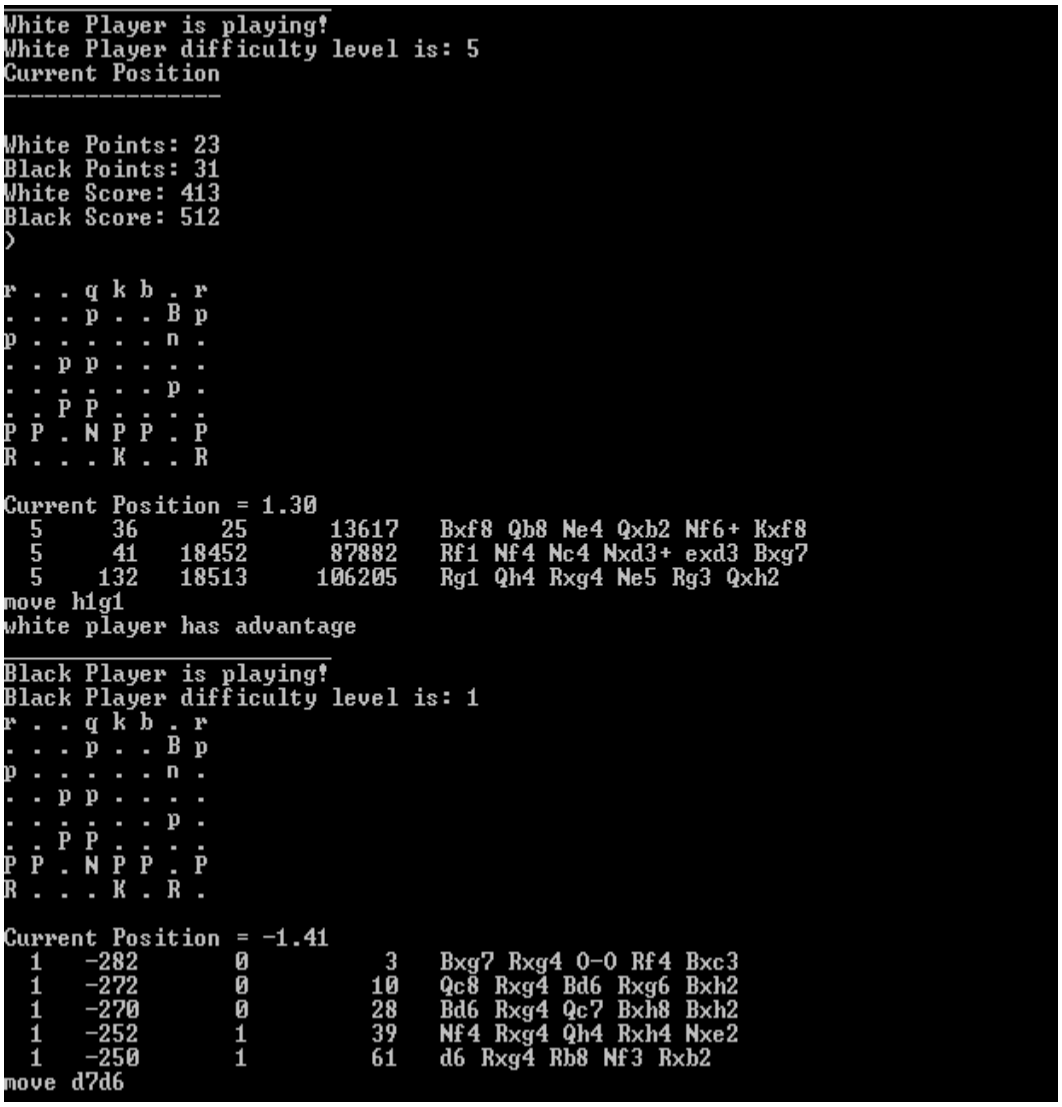

Figure 23.Screenshot 10 Test Case 2

After several moves, the current position for the white player changes to 2.05 and for the black player it changes to -3.66. Since the white player had an advantage over the black player (his current position was more than 1.3) for 3 consecutive rounds, the skill level for the white player changes to 4. The white

player makes the move b3c5 and the black player makes the move d6c5. You can see this in Figure 24.

Figure 24.Screenshot 11 Test Case 2

As shown in Figure 25, the white player's skill level is 4 and the black player's skill level is 1. The white player makes the move g4f4 and the black player makes the move f8g7. The white player's current position is 3.34 and the black player's current position is -3.58. Since the white player's current position is still greater than 1, he is in a better position than the black player.

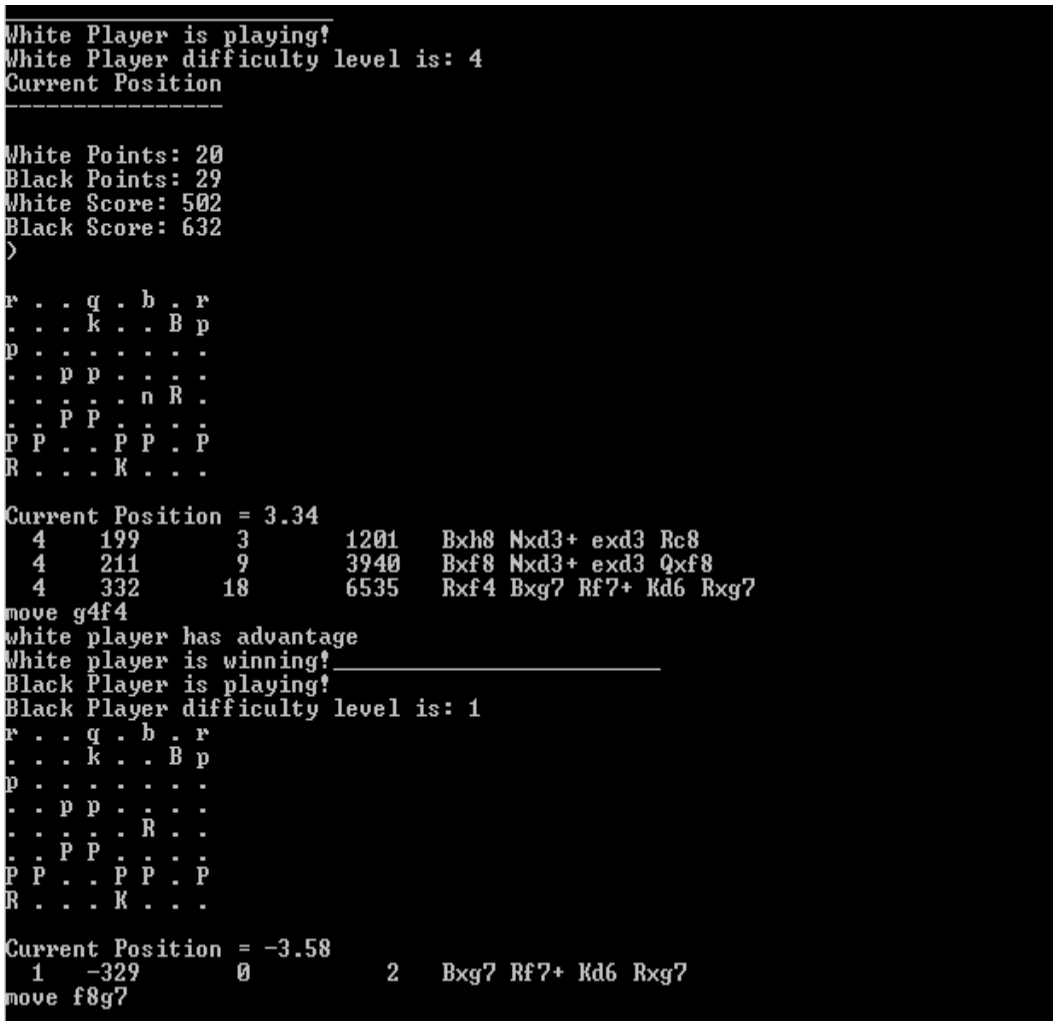

Figure 25.Screenshot 12 Test Case 2

The white player makes the move f4f5 and the black player makes the move g7c3. The current position for the white player is 3.35 and for the black player it is -3.3. This shows that the white player still has advantage over the black player. The skill level for the white player is 4 and for the black player it is 1. This is shown in Figure 26.

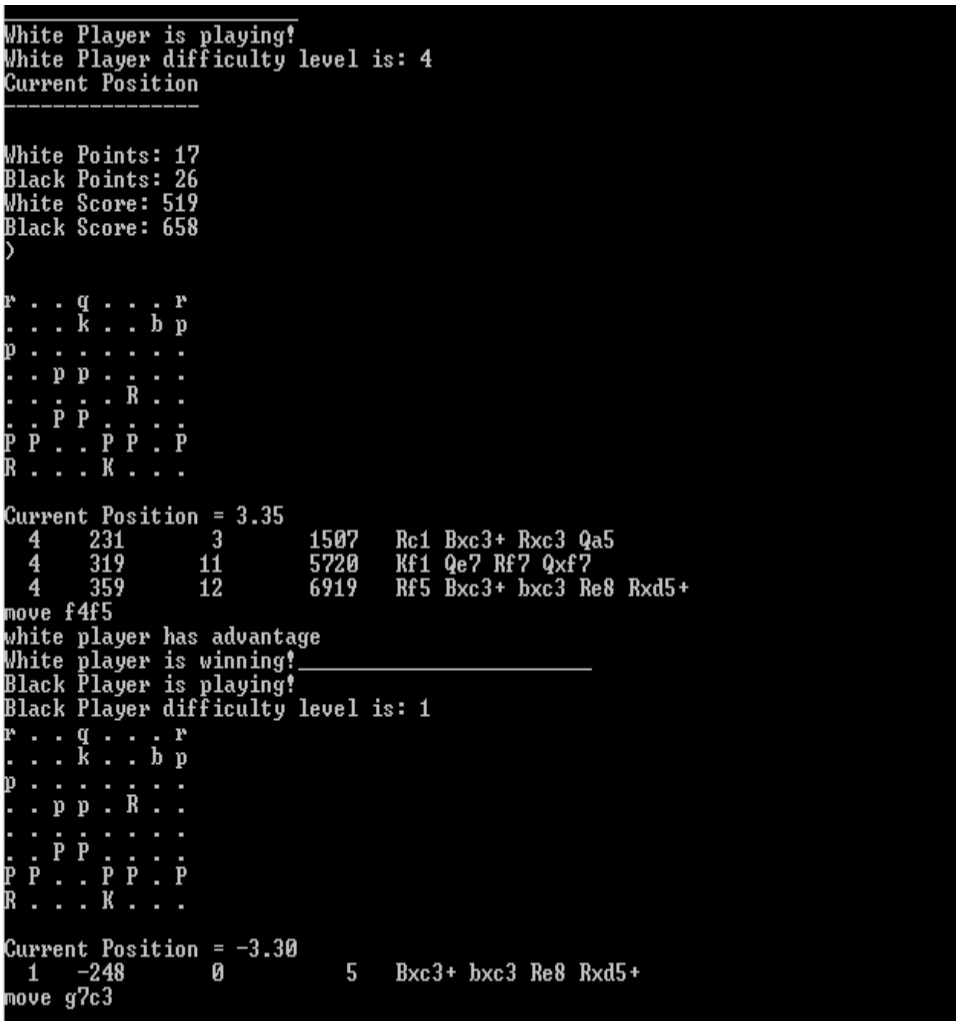

Figure 26.Screenshot 13 Test Case 2

The white player makes the move e1d1 and the black player makes the move c3e5. As shown in Figure 27, since the white player had advantage over the black player (his current position was more than 1.3) for 3 consecutive rounds, the skill level for the white player changes to 3. The black player still has a skill level of 1. The current position for the white player is 2.23 and for the black player it is -2.13.

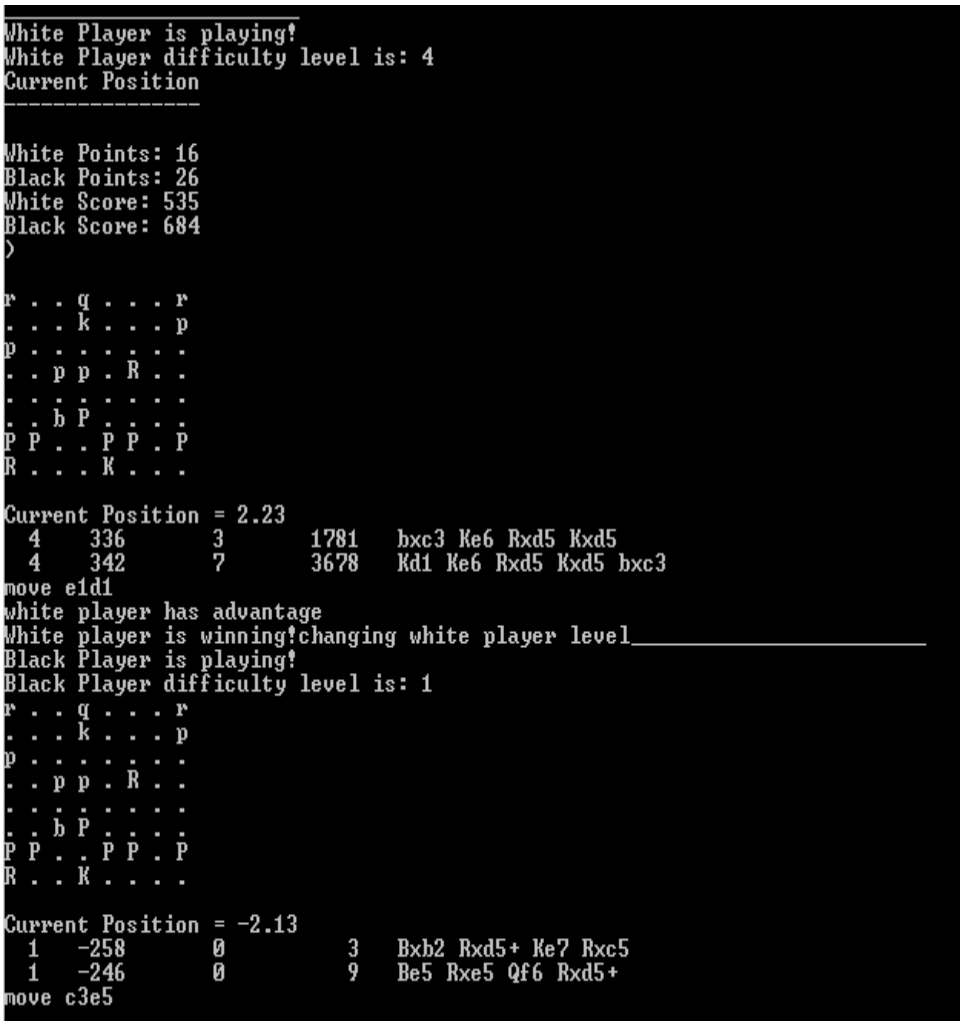

Figure 27.Screenshot 14 Test Case 2

The white player makes the move f5e5 and the black player makes the move a8c8. Now the white player's skill level is 3 and the black player's skill level is 1. The white player's current position is 2.12 and the black player's current position is -2.46. The white player's current position still shows that he has advantage over the black player. You can see this in Figure 28.

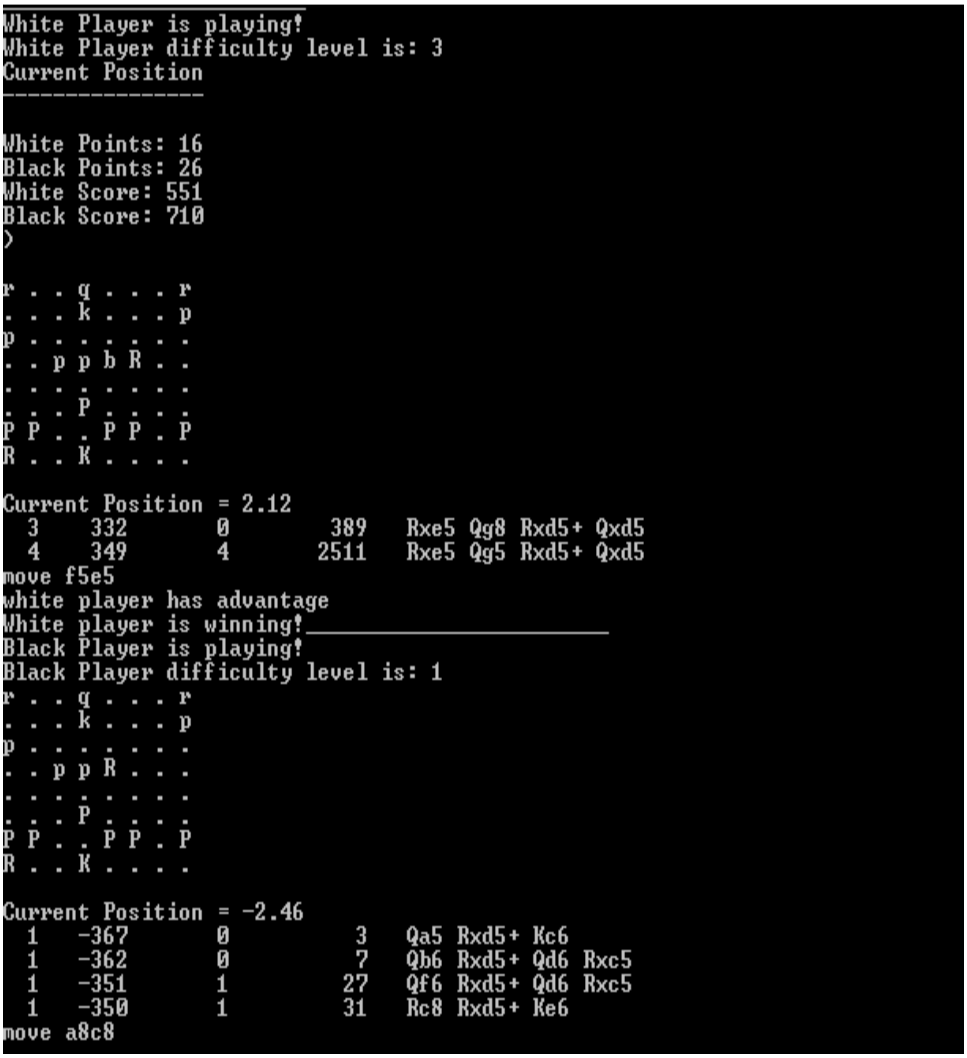

Figure 28.Screenshot 15 Test Case 2

After several moves, the white player makes the move a1c1 and the black player makes the move c6d5. The white player's current position changes to 3.69 and the black player's current position changes to -3.84. The white player is still in a better position than the black player. Since the skill level of the white player is more than 1.3 for 3 consecutive moves, the skill level for the white

player changes from 3 to 2. The black player still has a skill level of 1. This is shown in Figure 29.

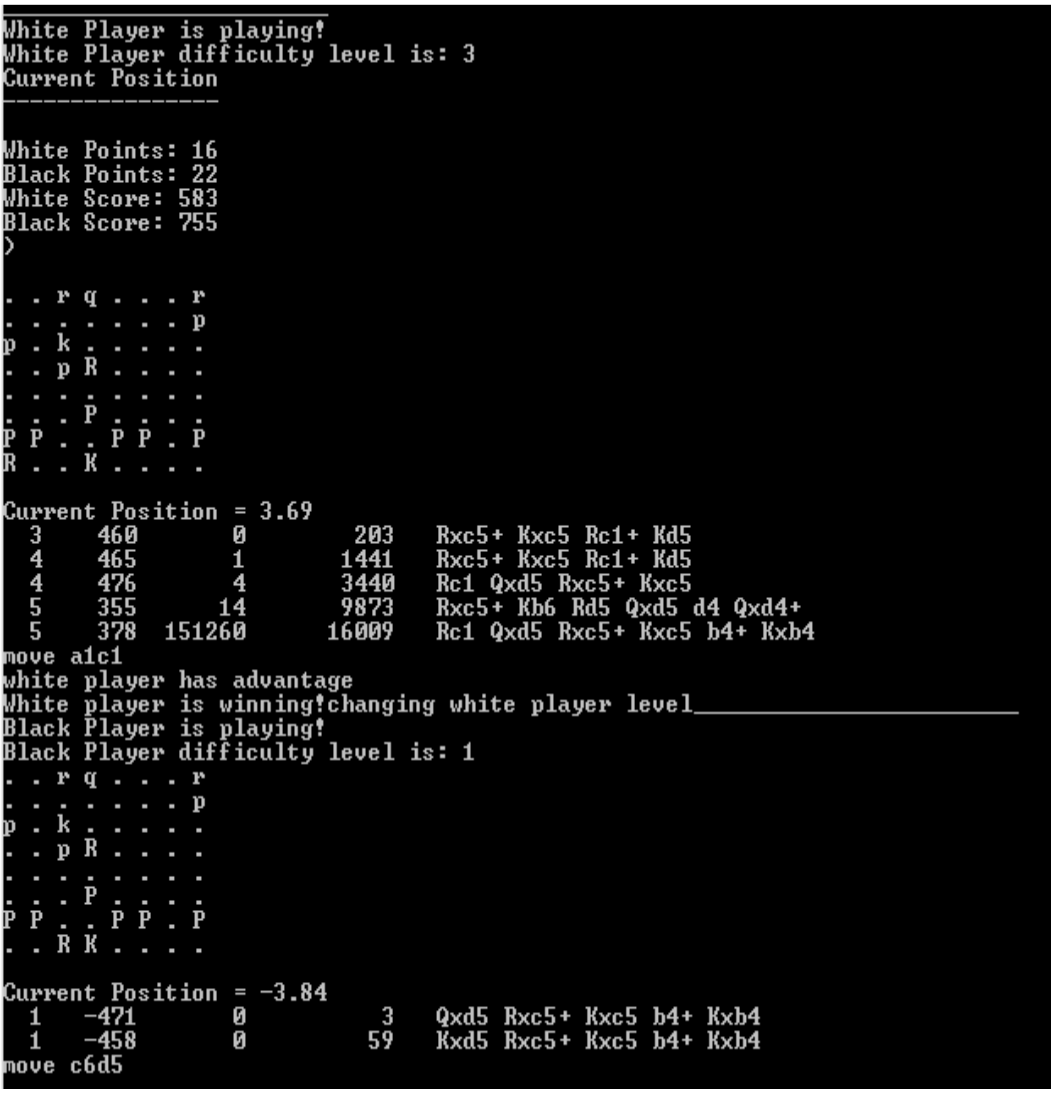

Figure 29.Screenshot 16 Test Case 2

The white player makes the move e2e4 and the black player makes the move d5e6. As shown in Figure 30, the white player's skill level is 2 and the

black player's skill level is 1. The current position for the white player changes to 3.27 and for the black player it changes to -3.35.

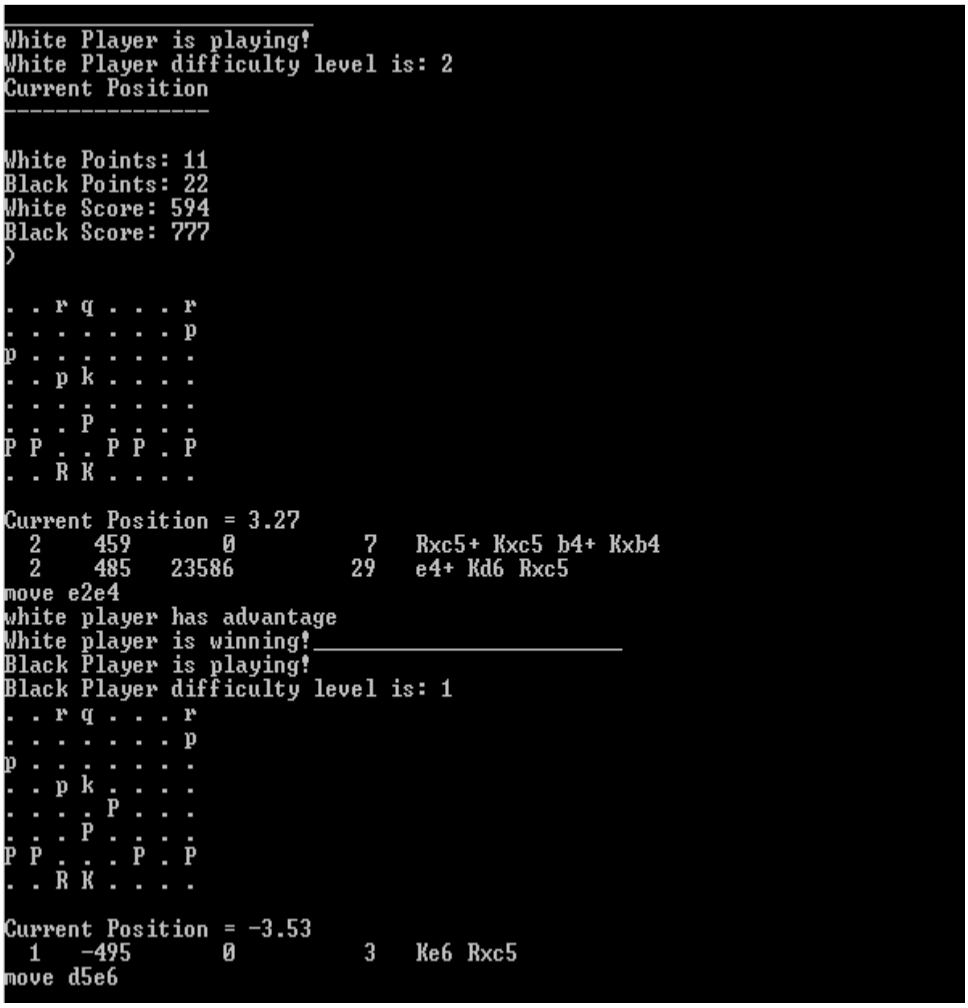

Figure 30.Screenshot 17 Test Case 2

The white player makes the move d1e2 and the black player makes the move d8d3. The current position for the white player changes to 3.8 and for the black player it changes to -4. The white player's skill level is 2 and the black

player's skill level is 1. We can see that the white player still has advantage over the black player. You can see this in Figure 31.

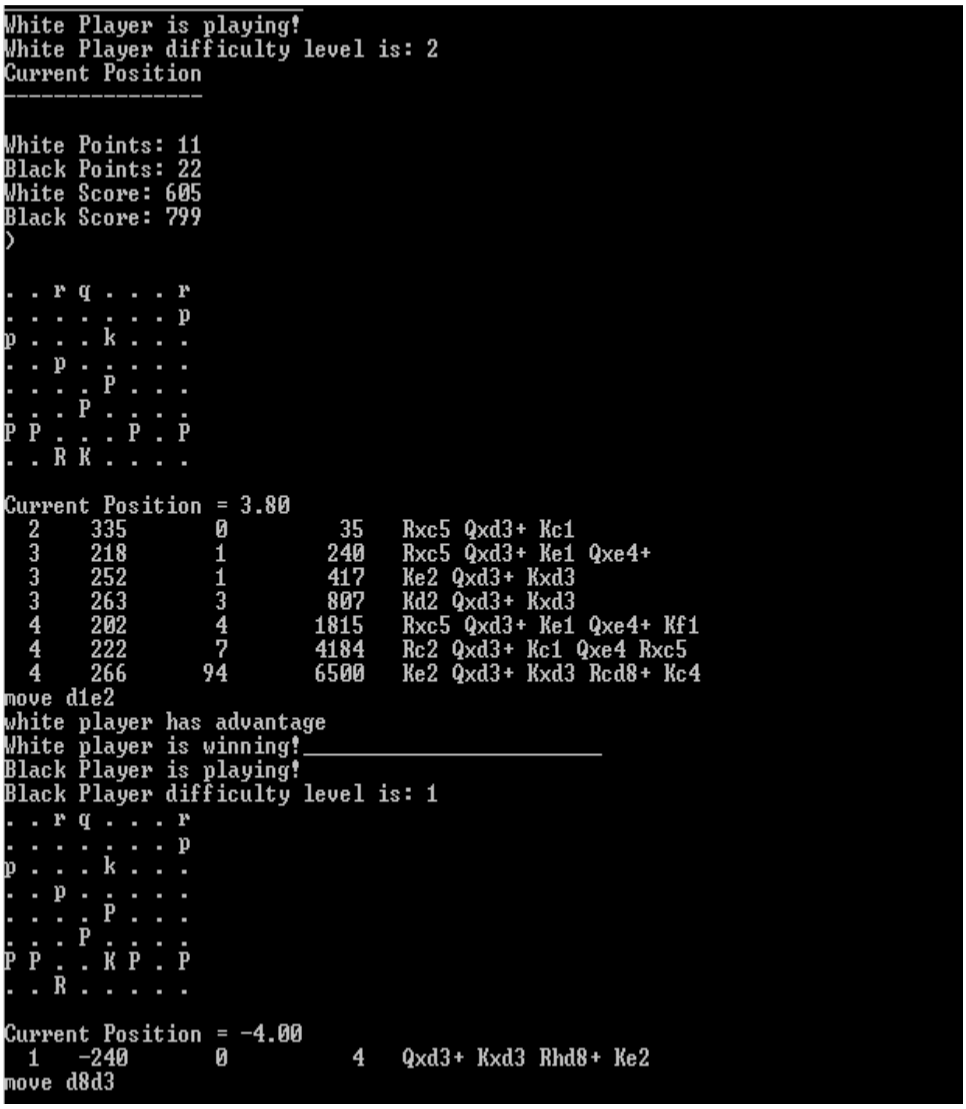

Figure 31.Screenshot 18 Test Case 2

Since the white player has had advantage over the black player for 3 consecutive moves, the skill level for the white player changes from 2 to 1. The black player still has a skill level of 1. The current position for the white player is

2.39 and for the black player it is -2.43. The white player makes the move e2d3 and the black player makes the move h8d8. This is shown in Figure 32.

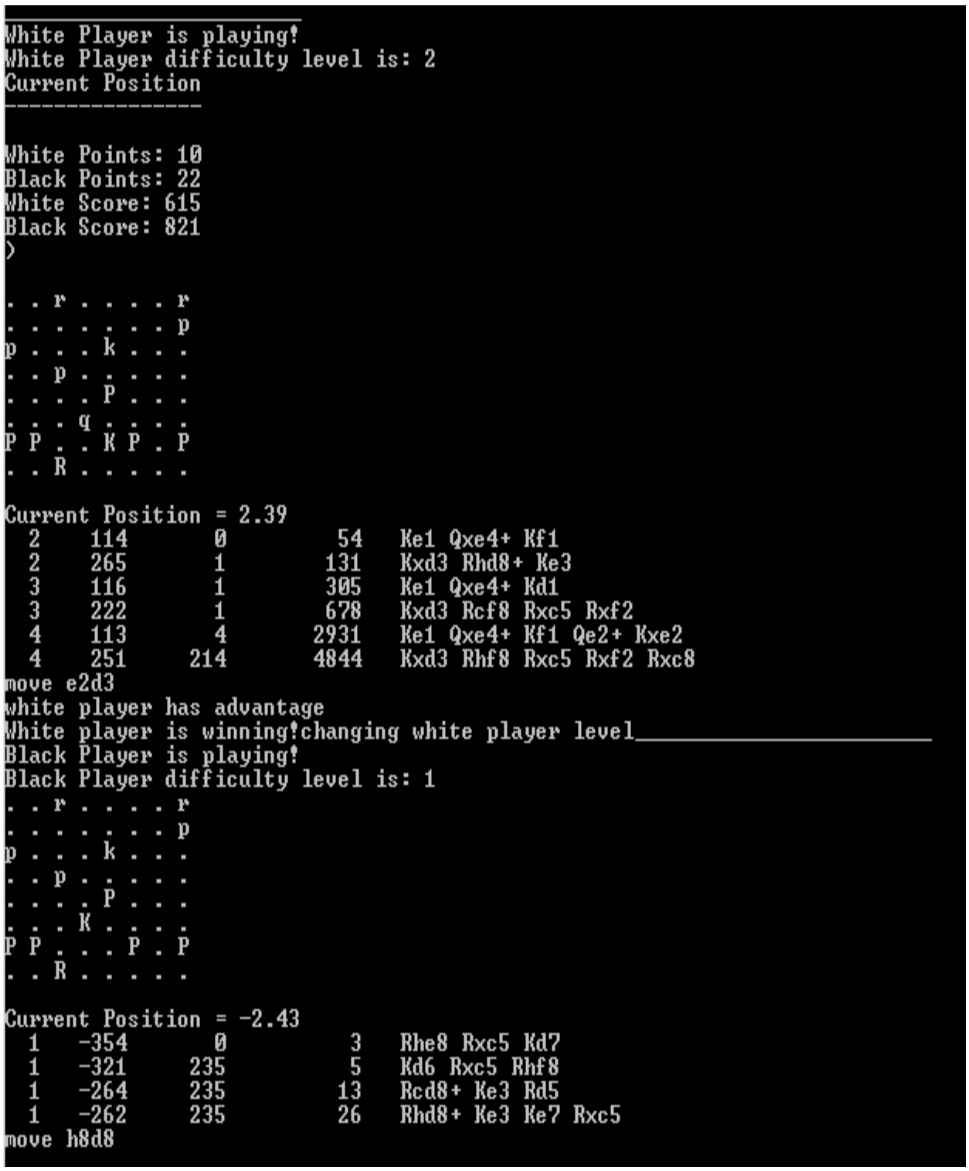

Figure 32.Screenshot 19 Test Case 2

Now both players have the same skill level. During this test case, the white player (adaptive engine) adapted to the black player. He started the game

with a skill level of 5 and his skill level kept adapting until his skill matched the black player's skill level of 1.

Table 2. Test Case 2

White Player's Move	Black Player's Move	White player's difficulty level	Black Player's difficulty level	White Player's current position	Black player's current Position	White Player's Score	Black Player's Score
C2C3	B8C6	5	1	0	0.11	39	39
D1B3	F7F6	5	1	-0.34	0.51	78	78
B3B7	C8B7	5	1	-0.47	-0.36	117	117
G1F3	E7E6	5	1	0.80	-0.98	147	155
F3G5	F6G5	5	1	1.01	-0.95	177	193
D2D3	G5G4	5	1	0.88	-1.17	204	231
C1E3	A7A6	5	1	1.16	-1.28	231	269
B1D2	G8E7	5	1	1.21	-1.40	258	307
G2D5	E6D5	5	1	1.48	-1.49	390	481
H1G1	D7D6	5	1	1.30	-1.41	413	512
B3C5	D6C5	5	1	2.05	-3.66	489	603
G4F4	F8G7	4	1	3.34	-3.58	502	632
F4F5	G7C3	4	1	3.35	-3.30	519	658
E1D1	C3E5	4	1	2.23	-2.13	535	684
F5E5	A8C8	3	1	2.12	-2.46	551	710

A1C1	**C6D5**	**3**	**1**	**3.69**	**-3.84**	**583**	**755**
E2E4	**D5E6**	**2**	**1**	**3.27**	**-3.53**	**594**	**777**
D1E2	**D8D3**	**2**	**1**	**3.80**	**-4**	**605**	**799**
E2D3	**H8D8**	**1**	**1**	**2.39**	**-2.43**	**615**	**821**

4.1.3. Test Case 3

In the third test case, the computer is not used to simulate the human player. The real human player (the white player) plays with a computer player (the black player). The objective of this test case is to test whether the black player (adaptive engine) is able to adapt to the real human player. The initial skill level for the black player (adaptive engine) is 5. The following screenshots show how this adaptation happens. When the game starts, the computer prompts the user to enter the player's name.

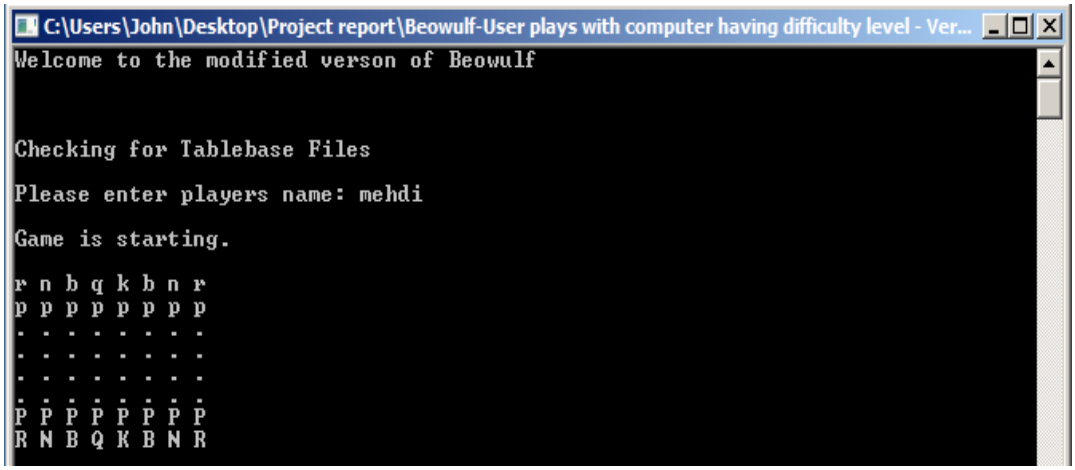

Figure 33.Game Starts Test Case 3

After the user enters his name, the game starts. The human player makes the first move: a2a3. As shown in Figure 34 the black player (adaptive engine) makes the move e7e6. The black player's difficulty level is 5 and the current position is -0.01.

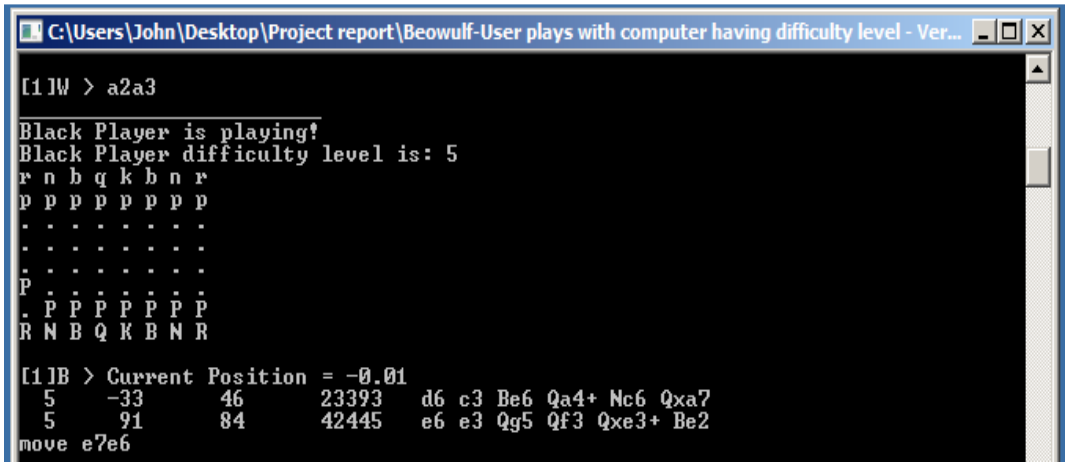

Figure 34.Screenshot 1 Test Case 3

As shown in Figure 35, after the human player makes the move c2c3 and the black player makes the move d8h4. The black player's current position changes to 0.39

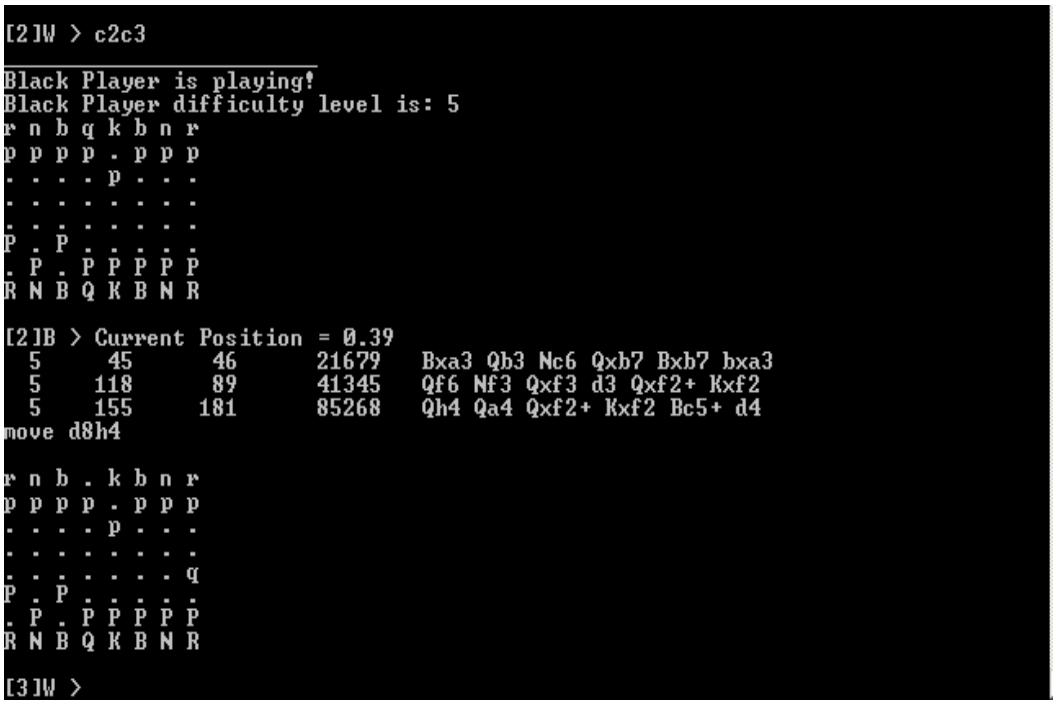

Figure 35.Screenshot 2 Test Case 3

Then the human player makes the move e2e3 and the black player makes the move h4f2. The current position for the black player is 0.03.

Figure 36.Screenshot 3 Test Case 3

As shown in Figure 37, the human player makes the move f2e1 and the black player makes the move f8a3. The black player's difficulty level is 5 and the current position for the black player is 1.49. According to section 3.3 since the black player's position is more than 1, he has advantage over the human player.

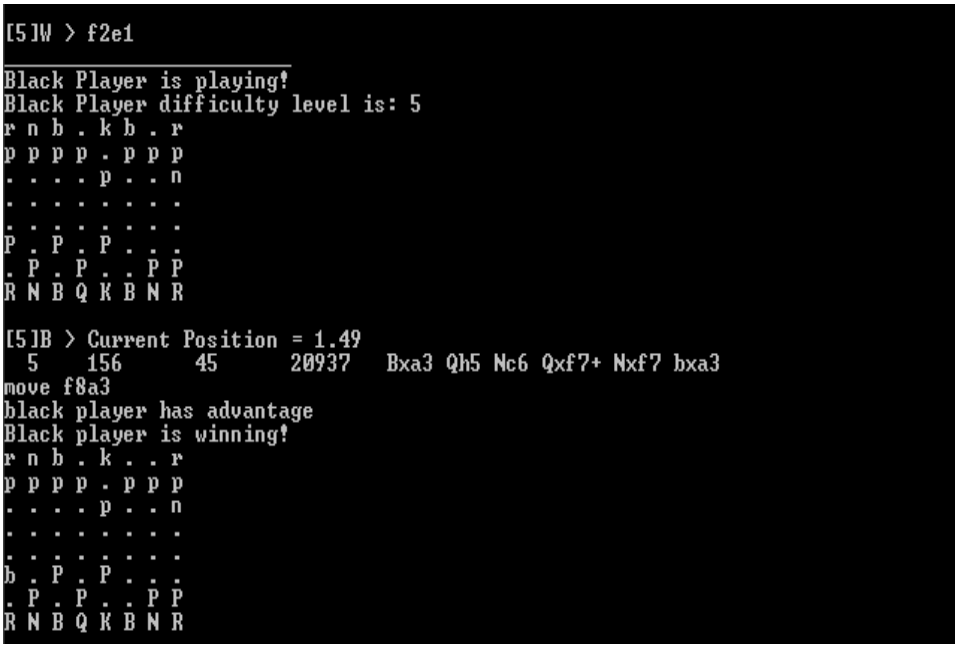

Figure 37.Screenshot 4 Test Case 3

As shown in Figure 38, the human player makes the move e1f2 and the black player makes the move g8h6. The current position for the black player is 1.42. The black player has advantage over the white player.

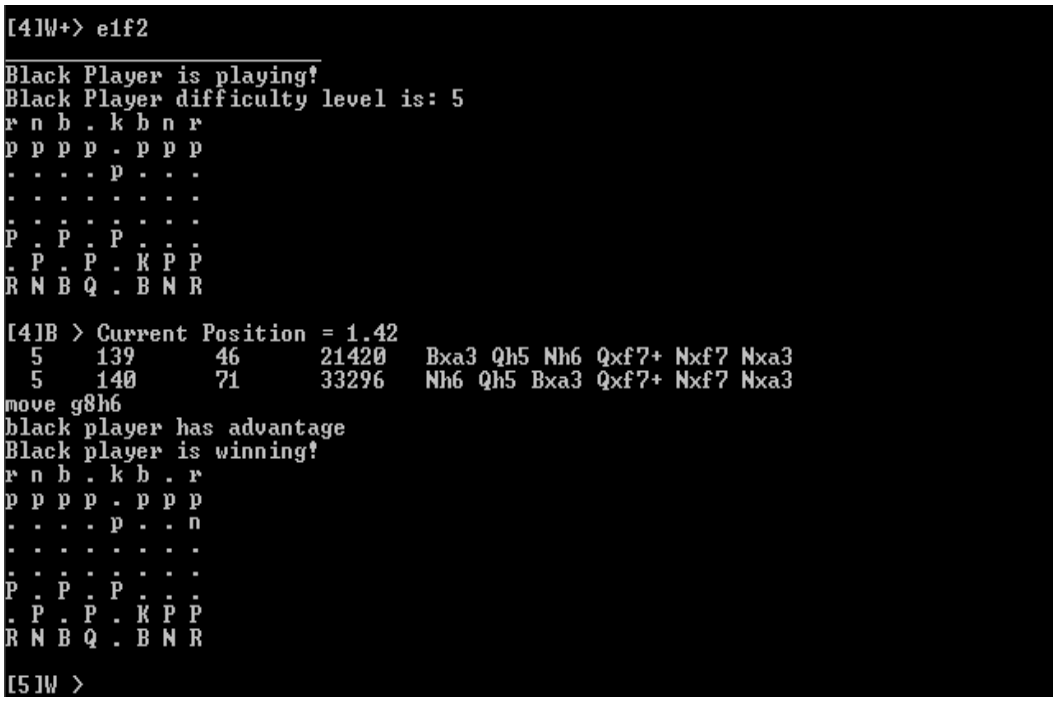

Figure 38.Screenshot 5 Test Case 3

As shown in Figure 39, the human player makes the move b2a3. The black player makes the move h6f5. The black player's difficulty level is 5 and its

current position is 2.7. The black player has advantage over the human player.

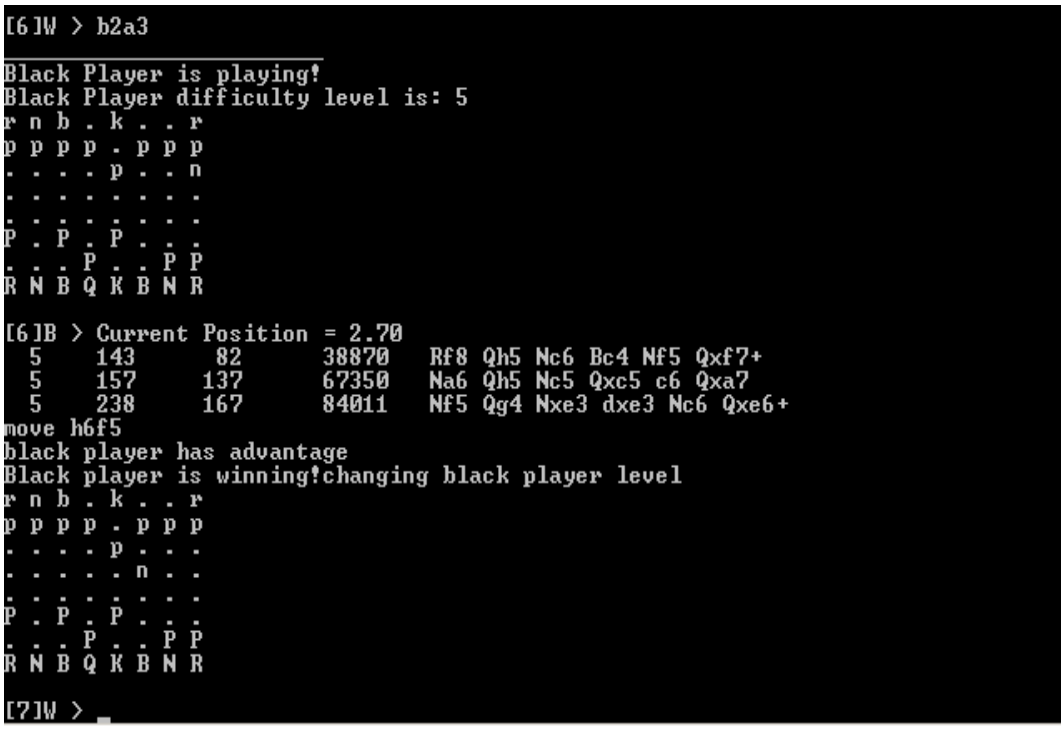

Figure 39.Screenshot 6 Test Case 3

As shown in Figure 40, the human player makes the move a3a4. The black player makes the move f5e3. The difficulty level for the black player is now 4 and its current position is 2.73. The black player still has advantage over the

white player and is winning the game.

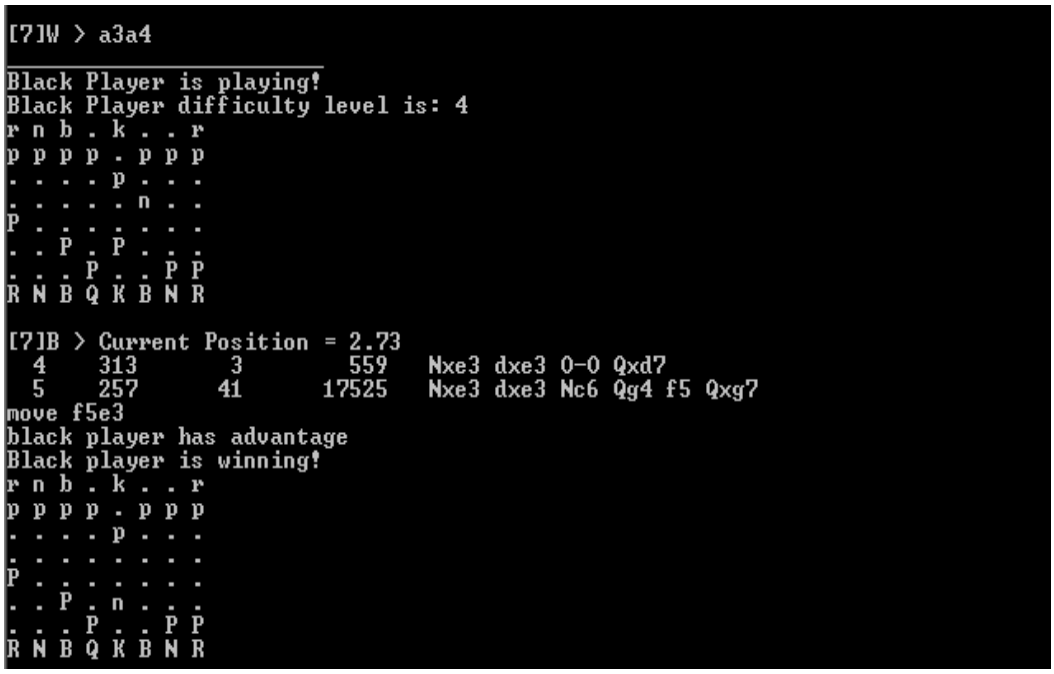

Figure 40.Screenshot 7 Test Case 3

As shown in Figure 41, the human player makes the move d2e3. The black player makes the move d7d5. The black player's difficulty level is 4 and its current position is 3.77. The black player has advantage over the white player and is winning the game.

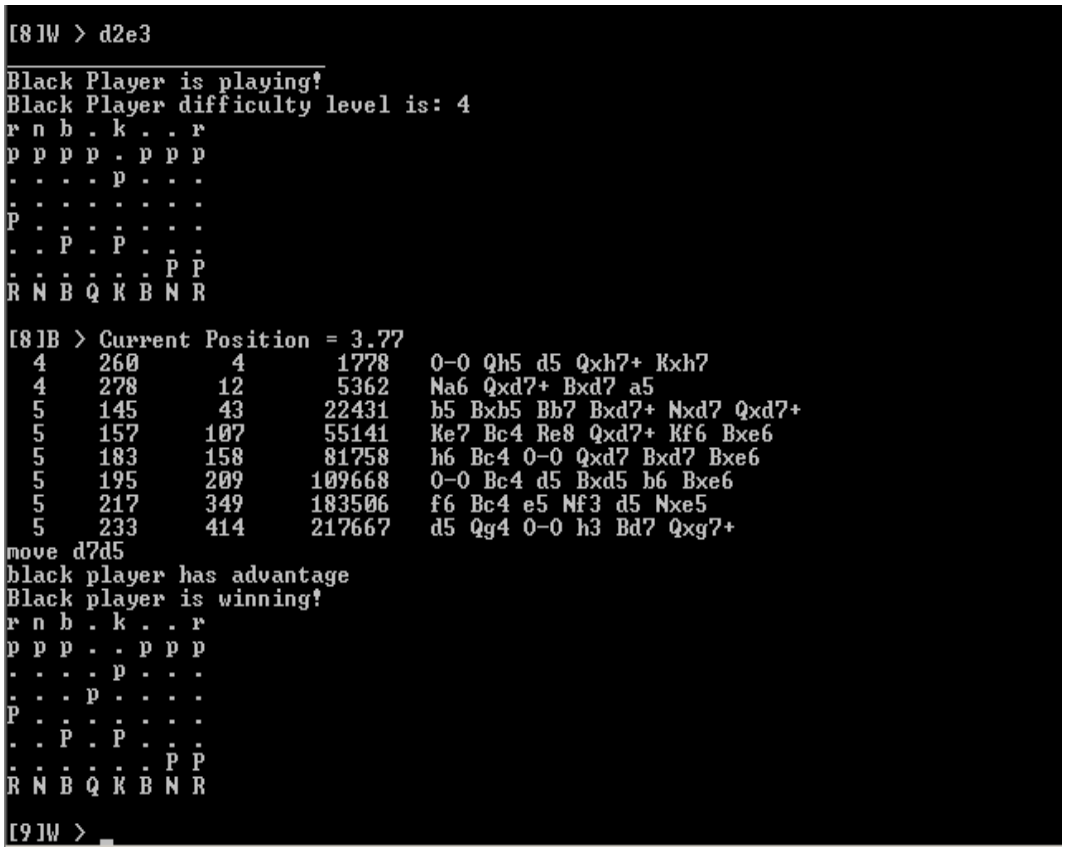

Figure 41.Screenshot 8 Test Case 3

As shown in Figure 42, the human player makes the move g2g3. The black player makes the move c8d7. The black player's difficulty level is 4 and its current position is 4.10. The black player has advantage over the white player

and is winning the game.

Figure 42.Screenshot 9 Test Case 3

As shown in Figure 43, the human player makes the move a4a5. The black player makes the move b8c6. The black player's difficulty level has changed to 3. The current position for the black player (adaptive engine) is 4.04. The black player has advantage over the white player and is winning the game.

Figure 43.Screenshot 10 Test Case 3

As shown in Figure 44, the human player makes the move h2h3. The black player makes move c6a5. The current position for the black player is 4.31 and its skill level is 3. The black player has advantage over the white player and is winning the game.

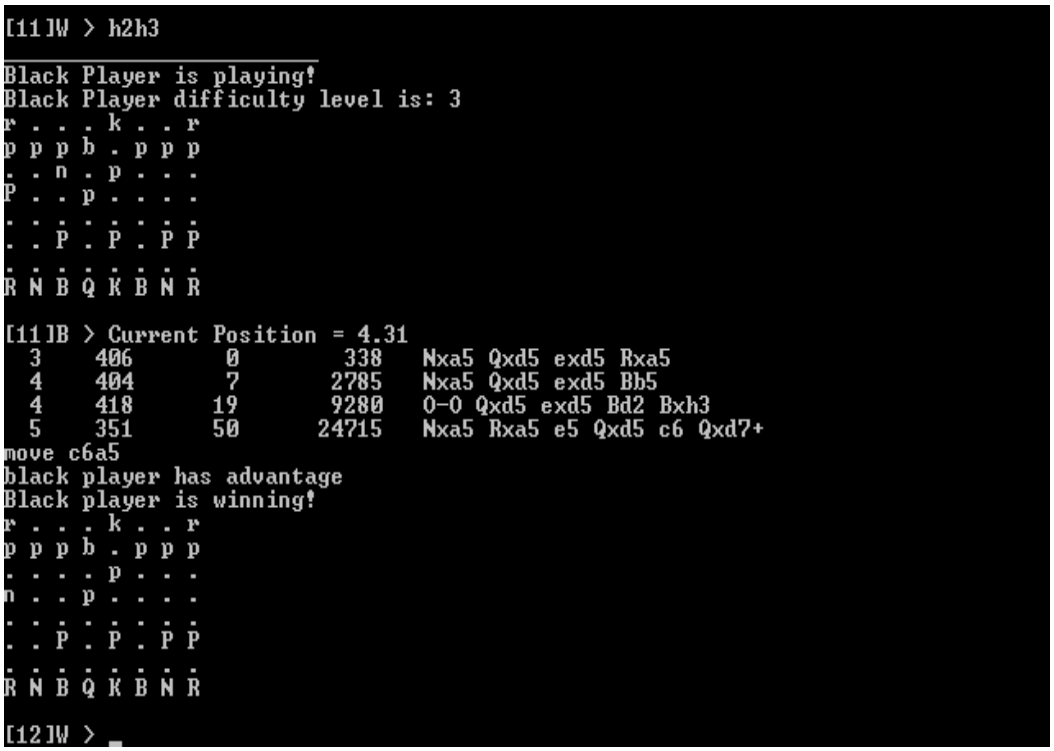

Figure 44.Screenshot 11 Test Case 3

As shown in Figure 45, the human player makes the move a4a5. The black player makes the move b8c6. The black player's difficulty level is 3. The current position for the black player is 4.04 and is winning the game.

Figure 45.Screenshot 12 Test Case 3

As shown in Figure 46, the human player makes the move g3g4. The black player makes the move o-o. The black player's difficulty level is 3. The current position for the black player is 5.54 and the black player is winning the game.

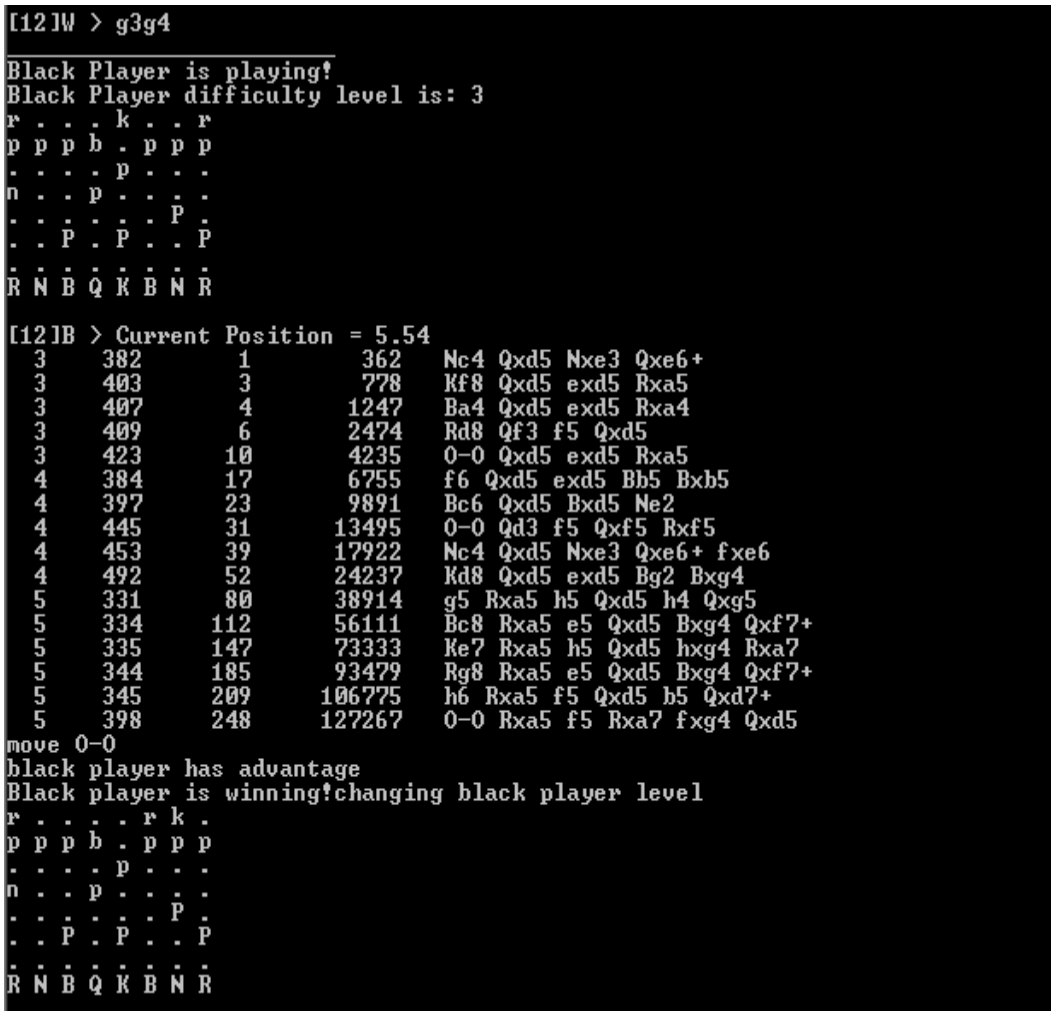

Figure 46.Screenshot 13 test Case 3

As shown in Figure 47, the human player makes the move g4g5. The black player makes the move f7f6. The black player's difficulty level changed to 2. The current position for the black player is 5.88. The black player has advantage over the human player and is winning the game.

Figure 47.Screenshot 14 Test Case 3

As shown in Figure 48, the human player makes the move g5f6. The black player makes the move f8f6. The black player's difficulty level is 2. The current position for the black player is 4.45 and the black player is winning the game.

Figure 48.Screenshot 15 Test Case 3

As shown in Figure 49, the human player makes the move h3h4. The black player makes the move f6f1. The black player's difficulty level is 2. The current position for the black player is 5.6. The black player has advantage over the white player and is winning the game.

Figure 49.Screenshot 16 Test Case 3

As shown in Figure 50, the human player makes the move e1f1. The black player makes the move d7b5. The black player's difficulty level changed to 1. The current position for the black player is 5.35. The black player has advantage over the white player and is winning the game.

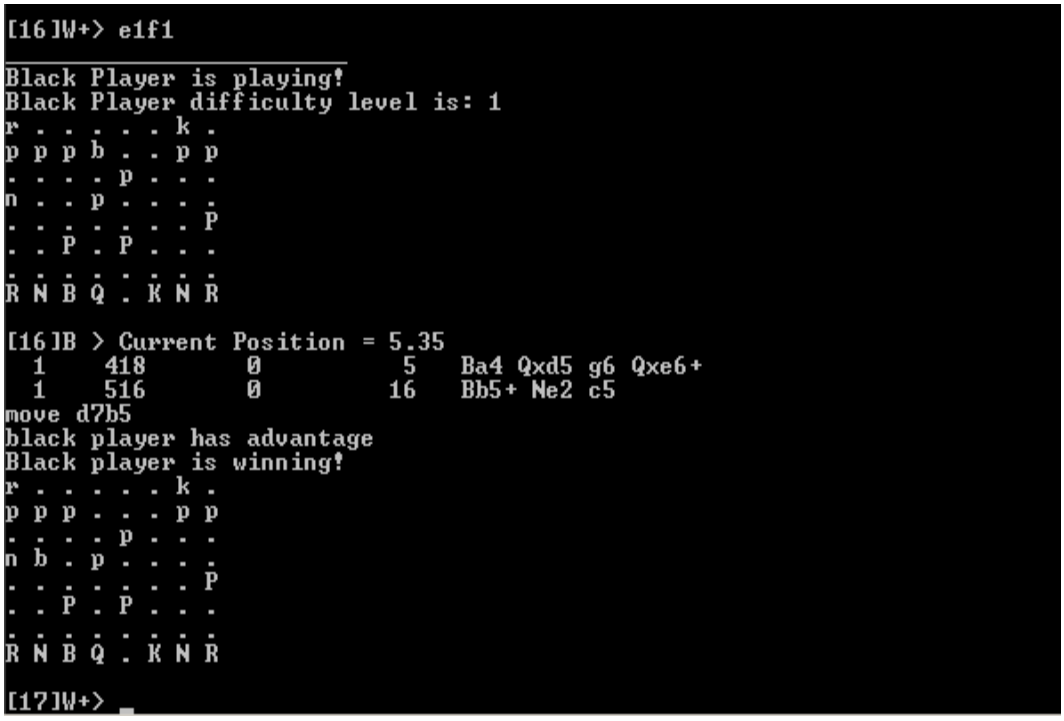

Figure 50.Screenshot 17 Test Case 3

As shown in Figure 51, the human player makes the move c3c4. The black player makes the move d5c4. The black player's difficulty level is 1 and the current position for the black player is 5.37. The black player has advantage and is winning the game.

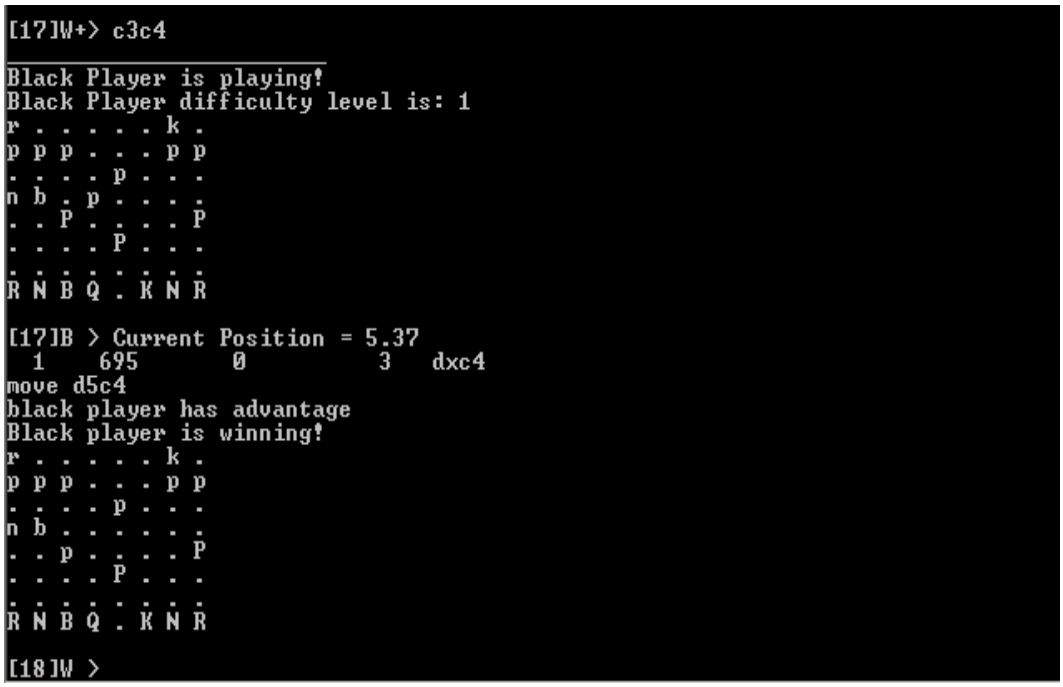

Figure 51.Screenshot 18 Test Case 3

Table 3. Test Case 3

White Player's Move	Black Player's Move	Black Player's difficulty level	Black Player's current position
A2A3	E7E6	5	-0.01
C2C3	D8H4	5	0.39
E2E3	H4F2	5	0.03
F2E1	F8A3	5	1.49
E1F2	G8H6	5	1.42
B2A3	H6F5	5	2.70

A3A4	**F5E3**	**4**	**2.73**
D2E3	**D7D5**	**4**	**3.77**
G2G3	**C8D7**	**4**	**4.10**
A4A5	**B8C6**	**3**	**4.04**
H2H3	**C6A5**	**3**	**4.31**
A4A5	**B8C6**	**3**	**4.04**
G3G4	**O-O**	**3**	**5.54**
G4G5	**F7F6**	**2**	**5.88**
G5F6	**F8F6**	**2**	**4.45**
H3H4	**F6F1**	**2**	**5.60**
E1F1	**D7B5**	**1**	**5.35**
C3C4	**D5C4**	**1**	**5.37**

As shown in table 3, the black player's (adaptive engine) initial skill level was 5 and it adapted to the human player's skill level. From this test case we can conclude that the human player's skill level was 1 during the game.

4.2. Results Analysis

We have 3 different test cases. In order to create the adaptive engine, we created another function that calculates points for each move and the game score for each player. This point system was not effective in evaluating the players' position in the game. Later, we didn't use that point system and instead we used the variable "Current Position" that is evaluated by the game engine.

In the first test case, computer played with another computer (white vs. black) and the white player adapted to the black player's skill level. The white player's initial skill level was 1 and the black player's initial skill level was 5. The white player took 9 moves to adapt from 1 to 5. The white player's position was 0 at the start of the game and it changed to -108.83 when the game ended. The black player's position was 0 at the start of the game, and it changed to 6.84 when the game ended. The black player won the game.

In test case 2, computer played with another computer (Black vs. White) and the black player adapted to the white player's skill level. The white player's initial skill level was 5 and the black player's initial skill level was 1.The black player took 19 moves to adapt to the white player's skill level. The white player's position was 0 when the game started, and it changed to 2.39 by the end of the game. The black player's position was 0.11 when the game started and it changed to -2.43 by the end of the game. The white player won the game.

In test case 3, a human player (white player) played with a computer (black player) and the computer adapted to the human player's level. The computer player took 18 moves to adapt to the human player. The black player's current position was -0.01 when the game started and it changed to 5.37 by the end of the game. The computer (black player) won the game. The computer's (black player) skill level was 1 after adaptation. We can conclude that the human player's skill level was 1 during the game.

CHAPTER FIVE

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 (Xboard or Winboard) for the adaptive chess engine, and creating a database to save each player's moves, current positions, and scores.

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_tree

[June 15, 2013].

7- "Minimax" Retrieved from:
https://en.wikipedia.org/wiki/Minimax
[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].