



Kabul Polytechnic University

Final Project

Intelligent Chess Game

Author :

Mir Omranudin Abhar

miromran@gamil.com

Supervisor :

Asst.Prof.Gholam Sakhi Shokouh

shokouh.sakhi@gmail.com

A final project submitted in fulfillment of the requirements
for the degree of bachelor
in the
Computer Engineering

November 15,2015

This page intentionally left blank.

Kabul Polytechnic University

Abstract

Faculty of Computer Engineering and Informatics
Department of Computer Engineering

Bachelor Degree

Intelligent Chess Game

by Mir Omranudin Abhar

Chess which do play between two players on a board is intellectual and mental game, it has its own rules of play which help to enhance and improve the mental and intellectual activities of the player, and this game has a huge amount of players around the all world they have strongly interested to have play it. This document deals with the fully computerized Chess Game, first, the game computerizes for two player to do play chess according all the valid rules of the chess on computer. Secondly, for making the game more interesting that will make users to direct do play against computer, computer intellectual force is added.

Acknowledgements

First of all, I would like to thank the most merciful and the most gracious Allah who teaches and shows us the way of happiness of this and next word by the agency of his messenger the owner of honor Mohammad (peace be open him) and give command for learning the knowledge. I am grateful from the all lectures especially my supervisor Asst.Prof.GHULAM SAKHI SHOKOH for his valuable contributions, patience and guidance through this study. I am grateful to my parents for their hidden help in terms of prayers and faith in me, which actually empowered me to fulfill this task. And last, I am grateful to all my entourage, classmates and companions who helped me on the journey of university career and made it easier.

I would also like to thank my entourage, classmates and companions especially from my partner Eng. Ismail Shenwari who helped me on the journey of university career and made it easier.

And last but not least, I am grateful to my parents for their hidden help in terms of prayers and faith in me, which actually empowered me to fulfill this task.

Contents

Abstract	i
Acknowledgment	iv
1 Introduction	2
1.1 introduction	2
2 Chess Game Introduction	5
2.1 Introduction	5
2.2 Chess Game Introduction	5
2.2.1 Rules in Chess Game	8
2.3 Summary	13
3 Computerized Chess	14
3.1 Introduction	14
3.2 A Brief History of Computer Chess	14
3.3 What we need for Computerized Chess	14
3.3.1 Board Representation	15
3.3.2 Move Generation	16
3.4 Summary	16
4 Intelligent Chess Game	18
4.1 Introduction	18
4.2 The Concept of Artificial Intelligence	18
4.3 Complexity intelligent chess game	20
4.4 What we needs for intelligent chess game	22
4.4.1 Search Techniques	23
4.4.2 Evaluation	23
4.5 Algorithms	24
4.5.1 MiniMax	24
4.5.2 Alpha-Beta	26
4.6 Summary	28
5 Development	29
5.1 Introduction	29
5.2 Design	29
5.3 Classes	29

5.3.1	MainClass.java Class	29
5.3.2	King.java Class	30
5.3.3	Queen.java Class	30
5.3.4	Knight.java Class	30
5.3.5	Bishop.java Class	30
5.3.6	Rook.java Class	30
5.3.7	Pawn.java Class	30
5.3.8	Algorithm.java Class	30
5.3.9	InterFace.java Class	31
5.4	Outline of the Work Done	31
5.5	Model	31
5.6	Summary	32
6	Implementation	36
6.1	Introduction	36
6.2	MainClass.java Class	36
6.3	King.java Class	37
6.4	Queen.java Class	37
6.5	Knight.java Class	37
6.6	Bishop.java Class	37
6.7	Rook.java Class	38
6.8	Pawn.java Class	38
6.9	InterFace.java class	38
6.10	Algorithm.java Class	38
7	Conclusion	40
7.1	Conclusion and future enhancement	40
7.2	Future Enhancement	40
	Bibliography	43
	Appendix A: Source Codes	47

List of Figures

1	Building Block of Computer Chess	3
2	History of Chess	5
3	Chess Board with pieces	7
4	Chessman Table	7
5	King Moves	8
6	Rook Moves	8
7	Bishop Moves	9
8	Queen Moves	9
9	Knight Moves	9
10	Pawn Moves	10
11	En-passant capturing	11
12	Pawn promotion	11
13	Castling	12
14	Check	12
15	checkmate	13
16	Stalemate	13
17	The Mechanical Turk	14
18	Playing steps(Human)	17
19	Artificial Intelligence	18
20	Chess Tree	20
21	Playing steps(Computer)	21
22	MiniMax Node	25
23	MinMax with Alpha Beta Algorithm	28
24	Chess Board with pieces	29
25	Classes in Chess Game	32
26	Environment Chess Game	33
27	Human Move	34
28	Computer Move	35

1 Introduction

Introduction This part describes an overview of whole project which provides an overarching theme of project to reader, but does not expand on specific details.

1.1 introduction

Chess is a game for two players, dubbed White and Black. The goal is to capture your opponent's king. In the game, this is known as a checkmate. Chess is played on a board with 64 squares. Each player begins with 16 pieces, lined up in two rows. The first row is occupied by pieces called pawns. The next row contains: a king, a queen, two rooks, two bishops, and two knights.

Chess is defined as a game of “perfect information”, because both players are aware of the entire state of the game world at all times: just by looking at the board, you can see which pieces are alive and where they are located. Checkers, Go, Go-Moku, Backgammon and Othello are other members of the category, but stud poker is not (you don't know what cards your opponent is holding in his hands).

Here that to able to change chess game from physical form to figurative form fully realistic, Several things are needed to make chess game computerized and intelligent.

- Some way to represent a chess board in memory, so that it knows what the state of the game is.
- Rules to determine how to generate legal moves, so that it can play without cheating (and verify that its human opponent is not trying to pull a fast one on it !)
- A technique to choose the move to make amongst all legal possibilities, so that it can choose a move instead of being forced to pick one at random.
- A way to compare moves and positions, so that it makes intelligent choices.
- Some sort of user interface.

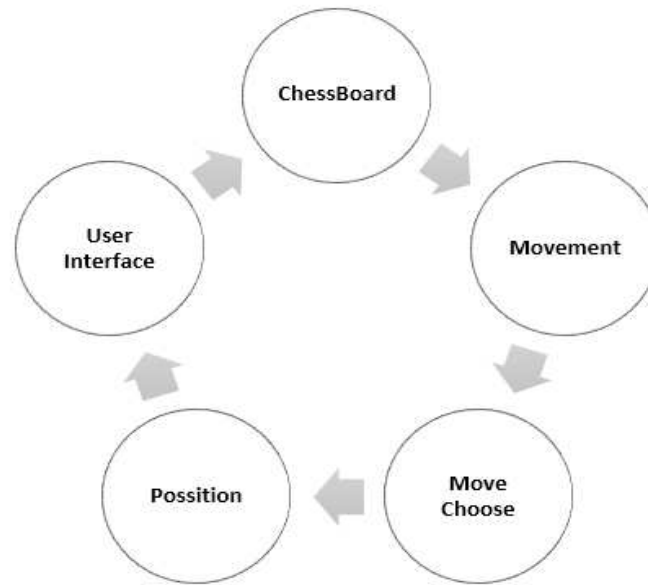


Figure 1: Building Block of Computer Chess

figure 1 illustrate Building Block of Computer Chess

After completing the whole implementation steps by one of computer programming language, thus we have been computerized an intelligent chess game which will able to act and take decision itself , and will able to play with human, the play will be real so the human will not fell that he is playing with computer, because computer will act and think like a human . we will discuss this issues with more details in this documentation.

The ultimate goal of this document is developing intelligent computer chess game by building a computerized chess game.

The scope of this project is limit for two players to play chess as real in computer. And a player also able to play with computer, and the computer able to play with its opponent in just one level according the intelligence which we gave it.

This document has been broken down into six chapters, chapter two describes the state and rules of related to chess game. The next chapter explores the methodology of used for computerizing the chess game. Chapter four explains the methodology of adding the intelligence to the game, chap-

ter five and six explain the development and implementation of chess game respectively while the last chapter dedicated to future enhancement and the summary of the entire document. Java has been selected for implementation of this project.

2 Chess Game Introduction

2.1 Introduction

In this section of document we pay all attention to introduce Chess fully detailed.

2.2 Chess Game Introduction

Chess is an old game. It likely came to the Western world from India in the 6th century A.D. Its a game, too, of royal origin: a test of someone's intelligence and acumen on the battlefield, yet today its everyone's game. For one, chess doesn't require expensive equipment to play. And the game has another virtue, which any player can explain: chess is fun.



Figure 2: History of Chess

In the Western world today, chess ranks among the most popular board games, and is played seemingly everywhere, by anyone: in urban parks, living rooms, schools, and well-publicized formal competitions. Children can learn to play chess at an early age, and by high school may young players reach a level of competence that approaches master-level play. Recently, some schools have begun incorporating chess play into their curricula. Chess may have a royal, and martial, origin, but in the modern world its a popular game of wits, enjoyed by many.

Technically, chess is a two-player board game comprising a formal system: a system of discrete tokens and rules for manipulating them. (In chess, there are six token types: King, Queen, Rook, Knight, Bishop, and Pawn. Each player begins the game with sixteen tokens selected from these types.) In

games-theory parlance, chess is a zero-sum, perfect information game. Zero-sum means that player A's successful move is to player B's detriment (a good move for A is a bad consequence for B), and perfect information means that all positions for each player are equally visible: the entire game at each step of play is perfectly visible to each player.

Chess also has what's called a "Markov" property, meaning that prior moves are unnecessary to understand how to play the next move. In principle, each discrete arrangement on the chess board can be viewed *sui generis*, and the next move can be determined by inspection of the current arrangement of game pieces. In computer science terms, chess is combinatorial, too: each successive move generates a (typically large) combination of possibilities. And, further, chess is a bounded branch problem in terms of search: computer scientists view chess as a large set of branching possibilities, bounded by poor moves on the "bottom" and good moves on the "top". The roundedness of chess lends itself to shortcuts, or "pruning", where additional possibilities can be ignored once one determines that a particular path along a branch is already poor, relative to another branch.

For all the games theory analysis, though, human players see chess in terms of tactics and strategy: thought. The ability for a human player to win at chess requires some degree of intellectual skill. One must "see" what is happening what the opponent moves signify in terms of a specific tactic, often part of a long-term strategy for victory. Still, the game at root remains entirely determined by the position of its game pieces, and the rules telling each player how they may be moved next.

The object of chess the goal of chess is to checkmate the opponent's king. Checkmate is when you attack the king, and the opponent cannot make a move which removes that attack. The player who is mated, loses the game.

The game is played on a squared board divided into 64 squares, alternating from light to dark. The board is always set up so that each player has the light square on her right-hand side, (Remember: light on the right). This figure 1 show the chess board with pieces. The Queen always stands on the square of

her own color. Thus, the light colored Queen must stand on the light colored square. A good way of remembering this is the saying: The Queen is a fashionable lady. She likes her dress to match her shoes! figure 3 illustrate Chess board with pieces

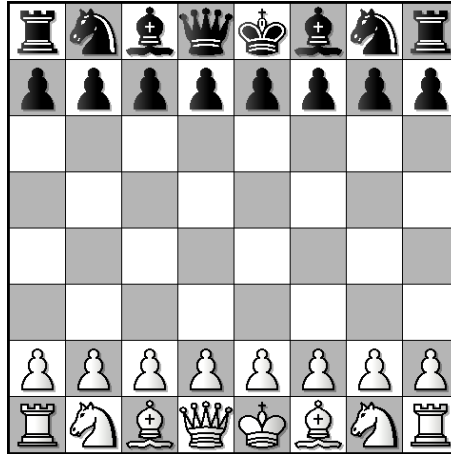


Figure 3: Chess Board with pieces

The pieces and pawns are called chessmen. They have different points to indicate how valuable they are. A Queen is worth 9 points so she is far more valuable than a pawn which is only worth 1 point. The King can never be captured and if he is in danger then he must move to a safe place or another chessman must make him safe. If he cannot reach safety then the game is lost. The person with the White pieces always begins the game. The image show below shows how the chessmen are represented in printed material, the symbol given to it and its value. figure 4 illustrate chessman table


Chessman	Name	Symbol	Value
	The King	K	Invaluable
	The Queen	Q	9 points
	The Rook	R	5 points
	The Bishop	B	3 points
	The Knight	N	3 points
	The Pawn	P	1 points

Figure 4: Chessman Table

2.2.1 Rules in Chess Game

Chess game has its own special rules, which cases to make it more interesting in the world.some of them listed bellow.

1. **King** The king can move horizontal, vertical and diagonal like the queen, but only one step at the time. The king may never enter a square which is threatened by an opposing piece. In other words you cannot place the king on a square when your opponent could capture the king next move. figure 5 illustrate King Moves.

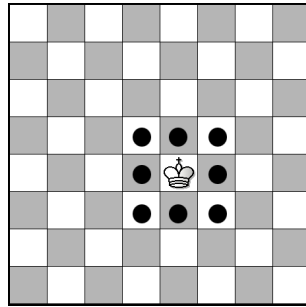


Figure 5: King Moves

2. **Rook** The rook moves vertical or horizontal in a straight line. The rook cannot jump over other pieces, all squares between the rook's current square and its destination must be empty. figure 6 illustrate Rook Moves.

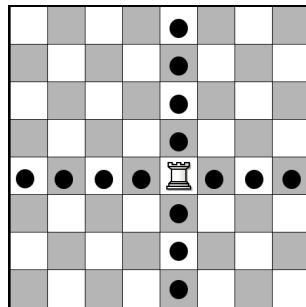


Figure 6: Rook Moves

3. **Bishop** The bishop moves diagonally in a straight line. Like the rook, the bishop cannot jump over other pieces. figure 7 illustrate Bishop Moves.
4. **Queen** The queen combines the movement of a rook and a bishop. This makes the queen the strongest piece on the board. The queen can move

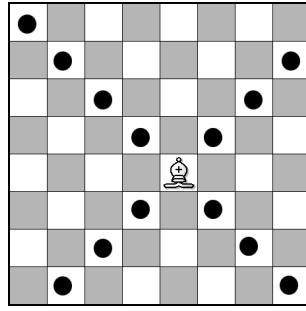


Figure 7: Bishop Moves

horizontal, vertical and diagonal in a straight line and may not jump over other pieces. figure 8 illustrate Queen Moves.

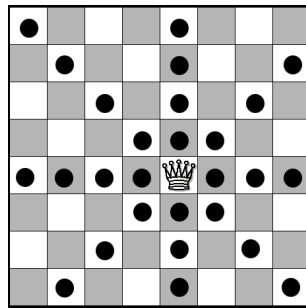


Figure 8: Queen Moves

5. **Knight** The knight makes a L-shaped move, which is a combination of 1 square horizontal or vertical, and one diagonal. The knight is the only piece that can jump over other pieces to travel from one square to another. The pieces the knight jumps over are not affected. figure 9 illustrate Knight Moves.

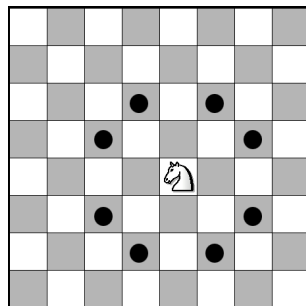


Figure 9: Knight Moves

6. **Pawn** The pawn can only move forward, but there are some variations in its movement which depend on the position of the pawn and the goal of the move. A pawn can always move one square forward, except when the destiny square is non-empty. For a pawn captures diagonally forward, and not straight ahead. Thus, a pawn can move one square diagonally, but only when capturing. There is another variation on the usual one step forward, only available to pawns which have not moved yet. From its starting position (second row for white, seventh row for black), a pawn can (not must) move two steps forward, if both squares in front of it are empty. figure 10 illustrate Pawn Moves.

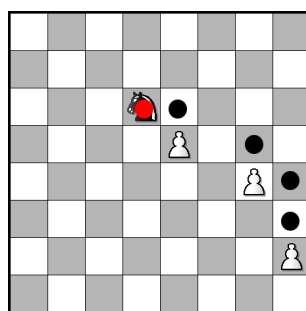


Figure 10: Pawn Moves

7. **En-passant capturing** A rule which is related to the pawn's double step is the en-passant capture. Move is only Available to a pawn, and only in this case: when a pawn takes a double step (from its starting position) and arrives on a square immediately next to an enemy pawn, this enemy pawn can capture en-passant. This means he captures the pawn as if it had moved just one step. You can make this move only immediately after the pawn's double step, not a move later. figure 11 illustrate En-passant capturing.
8. **Pawn promotion** when a player succeeds in getting a pawn on the other side of the board (White pawn on row 8 and black pawn on row 1), then this player must promote the pawn to a queen, rook, bishop or knight. By clicking one of the radio-buttons the pawn will change into the chosen piece. Most of the times a queen is chosen because it is the strongest piece. In this way, you can have up to nine queens! figure 12 illustrate Pawn promotion capturing.

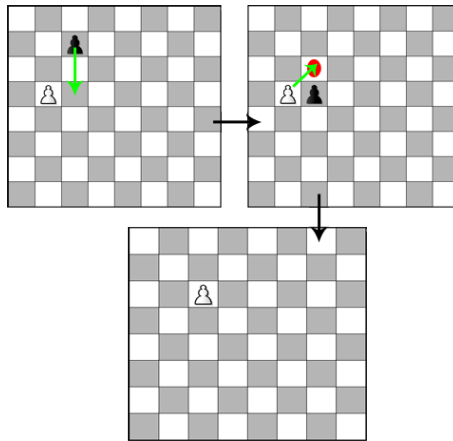


Figure 11: En-passant capturing

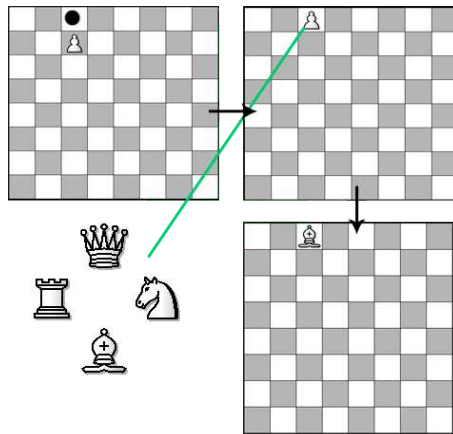


Figure 12: Pawn promotion

9. **Castling** Another special move is castling, This is the only move in the board game which allows you to Move two pieces, the king and a rook. You start castling by moving your king two squares to the left or to the right, Then the rook jumps over the king and arrives at the square the king just passed. The rook is automatically moved for you. figure 13 illustrate Pawn Castling.

Conditions necessary for castling:

- The king has not moved yet.
- The rook involved has not moved yet (the other rook doesn't matter).

- Every square between the king and the rook should be empty.
- The king is not in check.
- The king may not move over a square that is under attack by an enemy piece.
- After castling, the king may not be in check.

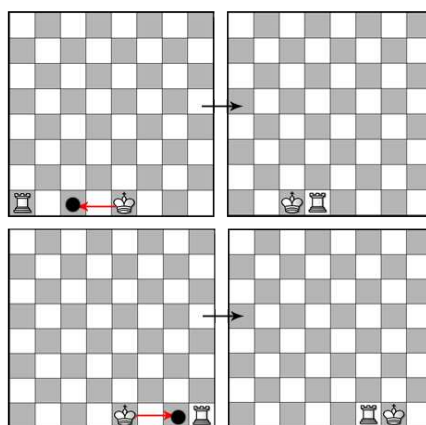


Figure 13: Castling

10. **Check** is a term used when a king can be captured by an enemy piece. A move which puts your king in check is illegal, and will not be accepted by the server. figure 14 illustrate Check.

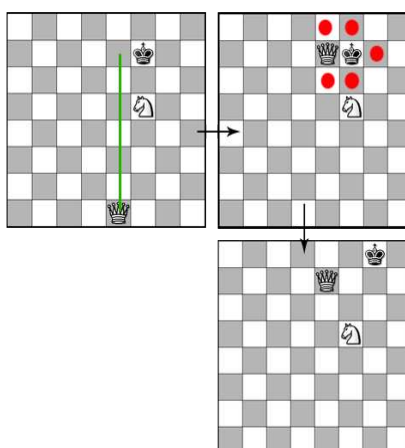


Figure 14: Check

11. **Checkmate** When a king is in check, and there is no move after which the king is not (again) in check, it is checkmate. The player who is mated loses the board game. figure 15 illustrate Checkmate.

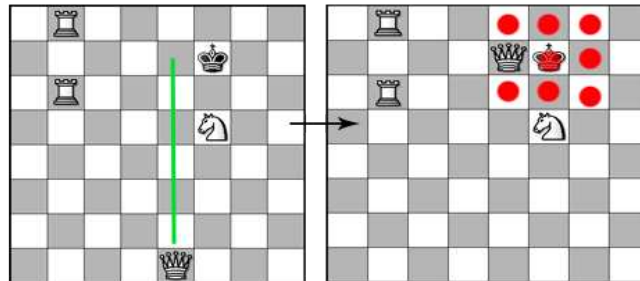


Figure 15: checkmate

12. **Stalemate** When a players king is not in check, but he has no legal moves (every move he can make would place his king in check) then it is stalemate and the board game is a draw. figure 16 illustrate Stalemate.

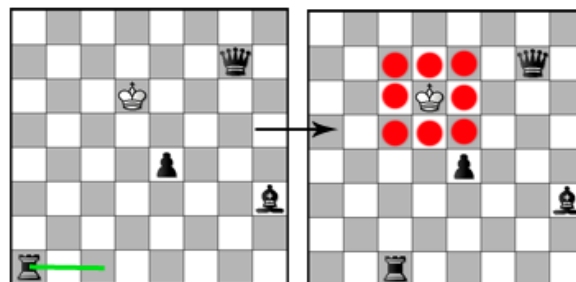


Figure 16: Stalemate

2.3 Summary

In this section of document chess game has been fully described with its all rules. In next Section we will computerize the game.

3 Computerized Chess

3.1 Introduction

In this section of document we like to computerize chess game. thus two players will able to direct play chess on computer.

3.2 A Brief History of Computer Chess

In 1770, diplomat and inventor, Wolfgang von Kempelen built a chess-playing machine called The Turk and presented it to the Empress Maria Theresa of Austria-Hungary. The Turk traveled to public fairs and royal courts alike for the next eighty-five years, playing such well known figures as Charles Babbage, Napoleon Bonaparte and Benjamin Franklin. figure 17 illustrate The Mechanical Turk.



Figure 17: The Mechanical Turk

In 1997, another man-made chess machine, a computer called Deep Blue, defeated the best chess player in the world. Although built more than two hundred years apart, both inventions were sensations in their day, appearing to be machines that could think. Although The Turk was ultimately revealed as a magic trick, Deep Blue was a computer based on advanced technology using powerful silicon chips and sophisticated software.

3.3 What we need for Computerized Chess

For making computerized chess game, which will let two players to play chess game in computer realistic. We need two things which we must introduce to

computer.

- Some way to represent a chess board in memory, so that it knows what the state of the game is.
- Rules to determine how to generate legal moves, so that it can play without cheating (and verify that its human opponent is not trying to pull a fast one on it !).

First we should introduce chess board and its elements to computer, next we need to produce movement for chess pieces according to rule if chess game.

3.3.1 Board Representation

In the early days of chess programming, memory was extremely limited (some programs ran in 8K or less) and the simplest, least expensive representations were the most effective. A typical chessboard was implemented as an 8x8 array, with each square represented by a single byte: an empty square was allocated value 0, a black king could be represented by the number 1, etc.

When chess programmers started working on 64-bit workstations and mainframes, more elaborate board representations based on “bit-boards” appeared. Apparently invented in the Soviet Union in the late 1960’s, the bit board is a 64-bit word containing information about one aspect of the game state, at a rate of 1 bit per square. Bit-boards are versatile and allow fast processing, because many operations that are repeated very often in the course of a chess game can be implemented as 1-cycle logic operations on bit-boards.

We used array data structure to represent Chess Board , small latters represent black pieces and capital latters represent white pieces.

```
chessBoard[8][8] = {
    {"r", "n", "b", "q", "k", "b", "n", "r"},
    {"p", "p", "p", "p", "p", "p", "p", "p"},
    {" ", " ", " ", " ", " ", " ", " ", " "},
    {" ", " ", " ", " ", " ", " ", " ", " "},
    {" ", " ", " ", " ", " ", " ", " ", " "},
    {"P", "P", "P", "P", "P", "P", "P", "P"},
    {"R", "N", "B", "Q", "K", "B", "N", "R"}
}
```

3.3.2 Move Generation

Once the board is represented we can shift to the move generator. Each piece in Chess has its own list of moves, therefore for each piece is necessary to write a specific function to generate the moves.

The rules of the game determine which moves (if any) the side to play is allowed to make. In some games, it is easy to look at the board and determine the legal moves: for example, in tic-tac-toe, any empty square is a legal move. For chess, however, things are more complicated: each piece has its own movement rules, pawns capture diagonally and move along a file, it is illegal to leave a king in check, and the “en-passant” captures, pawn promotions and castling moves require very specific conditions to be legal.

In fact, it turns out that move generation is one of the most computationally expensive and complicated aspects of chess programming. For generating movement in computer we need to define a function which will be called in each turn to identify and list the valid moves. Each move passes a bunch of steps

```
possibleMoves()
    list=null;
    for (i=0; i<64; i++)
        switch (chessboard[i/8][i%8])
            case "P":list =list+Pawn.possible(i);
            case "R":list =list+Rook.possible(i);
            case "B":list =list+Bishop.possible(i);
            case "N":list =list+Knight.possible(i);
            case "Q":list =list+Queen.possible(i);
            case "K":list =list+King.possible(i);
    return list;
```

before accomplish by a player in computerized chess game, we refer to represent by flow chart for better understanding. figure 18 illustrate this Flow Chart.

3.4 Summary

In this section after the representation of chess board and its element and pieces movement generation, we make able two players to play chess direct in computer. In next section we will give intelligence to the game.

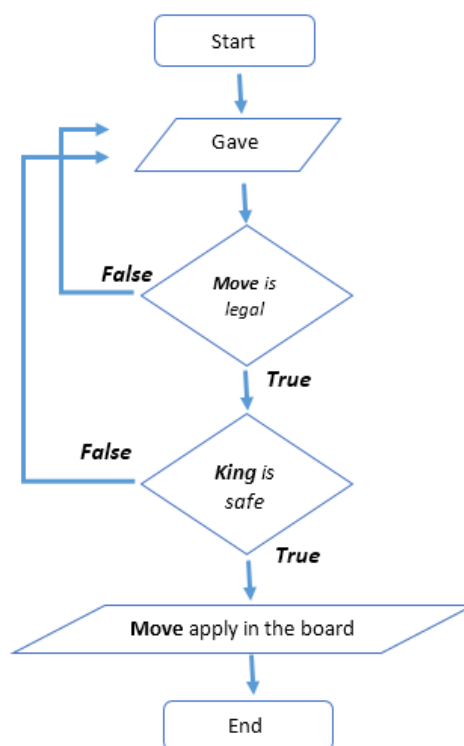


Figure 18: Playing steps(Human)

4 Intelligent Chess Game

4.1 Introduction

Chess game has been computerized in previous section, thus two players were able to directly play chess on computer. We will make a player to play chess with computer directly by adding intelligence to the game in this section.

4.2 The Concept of Artificial Intelligence

Artificial Intelligence (AI) is a branch of Science which deals with helping machines find solutions to complex problems in a more human-like fashion. This generally involves borrowing characteristics from human intelligence, and applying them as algorithms in a computer-friendly way. A more or less flexible or efficient approach can be taken depending on the requirements established, which influences how artificial the intelligent behavior appears.

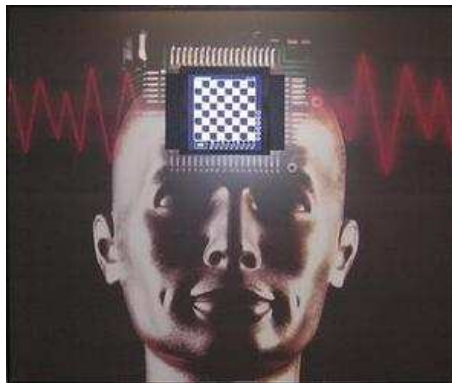


Figure 19: Artificial Intelligence

AI is generally associated with Computer Science, but it has many important links with other fields such as Maths, Psychology, Cognition, Biology and Philosophy, among many others. Our ability to combine knowledge from all these fields will ultimately benefit our progress in the quest of creating an intelligent artificial being. AI is one of the newest disciplines. It was formally initiated in 1956, when the name was coined, although at that point work had been under way for about five years. However, the study of intelligence is one of the oldest disciplines. For over 2000 years, philosophers have tried to understand how seeing, learning, remembering, and reasoning could, or should, be done. The

advent of usable computers in the early 1950s turned the learned but arm-chair speculation concerning these mental faculties into a real experimental and theoretical discipline. Many felt that the new “Electronic Super-Brains” had unlimited potential for intelligence. “Faster Than Einstein” was a typical headline.

But as well as providing a vehicle for creating artificially intelligent entities, the computer provides a tool for testing theories of intelligence, and many theories failed to withstand the test. AI has turned out to be more difficult than many at first imagined, and modern ideas are much richer, more subtle, and more interesting as a result.

AI currently encompasses a huge variety of subfields, from general-purpose areas such as perception and logical reasoning, to specific tasks such as playing chess, proving mathematical theorems, writing poetry, and diagnosing diseases. Often, scientists in other fields move gradually into artificial intelligence, where they find the tools and vocabulary to systematize and automate the intellectual tasks on which they have been working all their lives. Similarly, workers in AI can choose to apply their methods to any area of human intellectual endeavor. In this sense, it is truly a universal field.

Definition of Artificial intelligence

It is often difficult to construct a definition of a discipline that is satisfying to all of its practitioners. AI research encompasses a spectrum of related topics. Broadly, AI is the computer-based exploration of methods for solving challenging tasks that have traditionally depended on people for solution. Such tasks include complex logical inference, diagnosis, and visual recognition, comprehension of natural language, game playing, explanation, and planning. four possible goals to pursue in artificial intelligence:

- Systems that think like humans
- Systems that act like humans
- Systems that think rationally. (A system is rational if it does the right thing.)
- Systems that act rationally

4.3 Complexity intelligent chess game

For making the chess intelligent we should combat with a huge mount of complexity.

We start with a chessboard set up for the start of a game. Each player has 16 pieces. The white player starts the game all the time. At the beginning, white has 20 possible moves:

- The white player can move any pawn forward one or two positions.
- The white player can move either knight in two different ways.

The white player chooses one of these 20 moves and plays it. For the black player, the options are the same as 20 possible moves. So black chooses a move among those as well. Now white player can move again. This next move depends on the first move that white chose to make, but there are about 20 or so moves white can make given the current board position, and then black has 20 or so moves it can make, and so on. Actually the number of the moves both players can make usually increases as the game develops. This is how a computer program looks at chess. It thinks about it in a world of “all possible moves,” and it makes a very large tree (our search tree) for all of those moves. It can be visualized as follows:

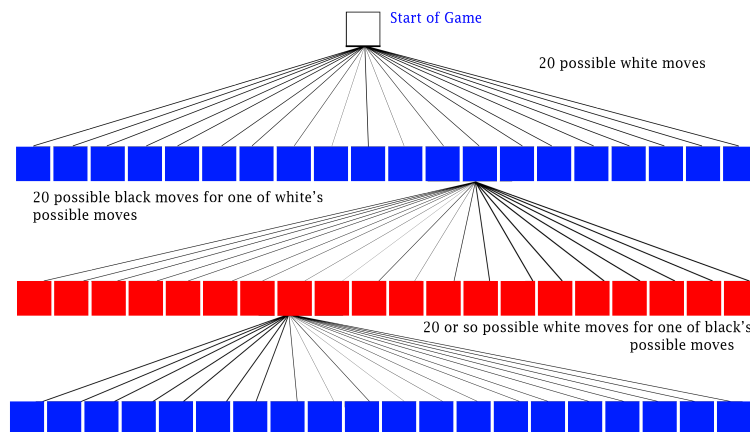


Figure 20: Chess Tree

A general computer chess program consists of three main parts:

1. **Move generator:** Generates all possible moves in a given position.

2. **Search function:** Looks at all possible moves and replies and try to find the best continuation.
3. **Position evaluator:** Gives a score to a position. It consists of a material, a mobility and a development score.

figure 21 illustrate steps Intelligence Chess Game. Here is the very general

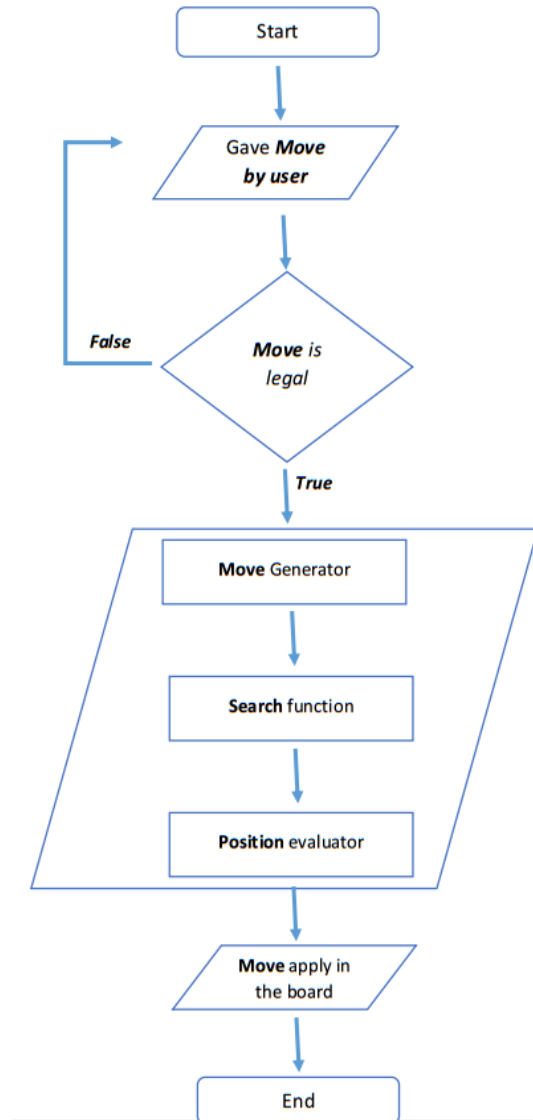


Figure 21: Playing steps(Computer)

pseudo code-like order for a computer chess program:

1. Get the input as an opponent move from the user (P: player).
 - Check whether this move is legal, prompt input again if not.

2. Generate a current board configuration based on the opponents (P) last move.
3. Set this current state as a root node for search tree.
4. Generate all possible moves for C (C: computer) as a response to this input.
 - Calculate the scores of each node using evaluation function.
5. For each of these moves, generate all possible user (P) responses.
 - Calculate the scores of each node using evaluation function.
6. As we now have the search tree of depth 2, apply Minimax search algorithm with Alpha-Beta Pruning to choose the best move for computer (C).
7. Output the computers move on the screen.
8. Check for situations like check, checkmate, draw etc. to end the game.
9. Repeat (1) to (8) until the game ends.

4.4 What we needs for intelligent chess game

We should introduce two things to computer for making the game intelligent, which will make the game to do an optimal move (a move which grant the most gain and give the most harm to opponent).

- A technique to choose the move to make amongst all legal possibilities, so that it can choose a move instead of being forced to pick one at random.
- A way to compare moves and positions, so that it makes intelligent choices

Chess game computerization needed two things a board representation an pieces movements generation as we discussed in previous section. For making the intelligent we need two other things as well, first legal moves creation and a random move selection, second move selection after evaluation of all generated moves intelligently.

4.4.1 Search Techniques

To a computer, it is far from obvious which of many legal moves are "good" and which are "bad". The best way to discriminate between the two is to look at their consequences (i.e., search series of moves, say 4 for each side and look at the results.) And to make sure that we make as few mistakes as possible, we will assume that the opponent is just as good as we are. This is the basic principle underlying the minimax search algorithm, which is at the root of all chess programs.

Unfortunately, minimax' complexity is $O(b^n)$, where b ("branching factor") is the number of legal moves available on average at any given time and n (the depth) is the number of "plies" you look ahead, where one ply is one move by one side. This number grows impossibly fast, so a considerable amount of work has been done to develop algorithms that minimize the effort expended on search for a given depth. Iterative-deepening Alphabeta, NegaScout and MTD(f) are among the most successful of these algorithms, we will discuss with more detail later in this document.

Another major source of headaches for chess programmers is the "horizon effect", first described by Hans Berliner. Suppose that your program searches to a depth of 8-ply, and that it discovers to its horror that the opponent will capture its queen at ply 6. Left to its own devices, the program will then proceed to throw its bishops to the wolves so that it will delay the queen capture to ply 10, which it can't see because its search ends at ply 8. From the program's point of view, the queen is "saved", because the capture is no longer visible... But it has lost a bishop, and the queen capture reappears during the next move's search. It turns out that finding a position where a program can reason correctly about the relative strength of the forces in presence is not a trivial task at all, and that searching every line of play to the same depth is tantamount to suicide. Numerous techniques have been developed to defeat the horizon effect.

4.4.2 Evaluation

Finally, the program must have some way of assessing whether a given position means that it is ahead or that it has lost the game. This evaluation depends

heavily upon the rules of the game: while "material balance" (i.e., the number and value of the pieces on the board) is the dominant factor in chess, because being ahead by as little as a single pawn can often guarantee a victory for a strong player, it is of no significance in Go-Moku and downright misleading in Othello, where you are often better off with fewer pieces on the board until the very last moment.

4.5 Algorithms

The ordered list of the sub-tasks which is used to perform the functional and conditional steps to solve a task is called Algorithm, or Algorithm is a set of commands that solve a task/problem step by step and should have the following conditions:

- It has to be precise and clear
- It has to be described with details
- The execution of the commands should be ordered
- The execution should have an end

Algorithm is used to divide a complicated function into smaller functions.

To give the intelligence to computer chess game which will form and make it able to think, take decision, do an optimal movement and compete against its opponents (human) automatically as real, in other words to make the chess game intelligent which will able to do optimal moves to gain the most profit and give the most damage to its opponent(human),for doing this task we need to implement Minimax with the Alpha Beta pruning algorithm which uses evaluation function to generate optimal moves and make the chess intelligent the relate to this issues discussed in this section.

4.5.1 MiniMax

The minimax algorithm is a way of finding an optimal move in a two player game. In the search tree for a two-player game, there are two kinds of nodes, nodes representing your moves and nodes representing your opponent's moves. Nodes representing your moves are generally drawn as squares (or possibly upward pointing triangles): These are also called MAX nodes. The goal at a

MAX node is to maximize the value of the subtree rooted at that node. To do this, a MAX node chooses the child with the greatest value, and that becomes the value of the MAX node. Nodes representing your opponent's moves are generally drawn as circles (or possibly as downward pointing triangles): These are also called MIN nodes. The goal at a MIN node is to minimize the value of the subtree rooted at that node. To do this, a MIN node chooses the child with the smallest value, and that becomes the value of the MIN node. figure 22 illustrate MiniMax Nodes.

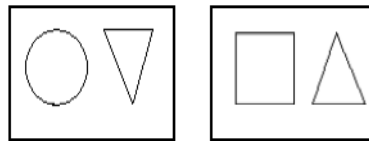


Figure 22: MiniMax Node

The Minimax: Assume that both White and Black plays the best moves. We maximizes Whites score, perform a depth-first search and evaluate the leaf nodes, Choose child node with highest value if it is White to move, Choose child node with lowest value if it is Black to move.

Minimax search method for chess can be defined as follows:

- Assume that there is a way to evaluate a board position so that we know whether Player 1 (Max) is going to win, whether his opponent (Min) will, or whether the position will lead to a draw. This evaluation takes the form of a number: a positive number indicates that Max is leading, a negative number, that Min is ahead, and a zero, that nobody has acquired an advantage.
- Max's job is to make moves, which will increase the board's evaluation.
- Min's job is to make moves, which decrease the board's evaluation.
- Assume that both players play in a way that they never make any mistakes and always make the moves that improve their respective positions the most.

The trouble with Minimax is that there is an exponential number of possible paths which must be examined. This means that effort grows dramatically with:

- The number of possible moves by each player, called the branching factor.
- The depth of the look-ahead, and usually described as "N-ply", where N is an integer number and "ply" means one move by one player. The simplest approach we would follow is searching to a depth of 2-ply, one move per player.

MiniMax Algorithm

```

function Max – Value(state) returns a utility value
  if Terminal – Test(state) then return Utility(state)
   $v \leftarrow -\infty$ 
  for  $a, s$  in Successors(state) do  $v \leftarrow \text{Max}(v, \text{Min – Value}(s))$ 
  return  $v$ 

function Min – Value(state) returns a utility value
  if Terminal – Test(state) then return Utility(state)
   $v \leftarrow +\infty$ 
  for  $a, s$  in Successors(state) do  $v \leftarrow \text{Min}(v, \text{Max – Value}(s))$ 
  return  $v$ 

```

4.5.2 Alpha-Beta

The minimax algorithm is a way of finding an optimal move in a two player game. Alpha-beta pruning is a way of finding the optimal minimax solution while avoiding searching sub trees of moves which won't be selected. In the search tree for a two-player game, there are two kinds of nodes, nodes representing your moves and nodes representing your opponent's moves.

The Alpha-Beta Pruning algorithm is a significant enhancement to the minimax search algorithm that eliminates the need to search large portions of the game tree. If one already has found a quite good move and search for alternatives, one refutation is enough to avoid it. No need to look for even stronger refutations. The algorithm maintains two values, alpha and beta. They represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of respectively.

Alpha-beta pruning gets its name from two bounds that are passed along during the calculation, which restrict the set of possible solutions based on the portion of the search tree that has already been seen. Specifically, α Alpha is

Alpha Beta Pruning Algorithm

```

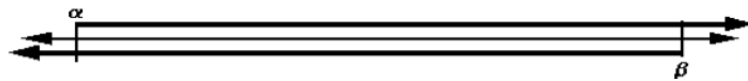
function Max – Value(state,  $\alpha$ ,  $\beta$ )
    if Terminal – Test(state) then return Utility(state)
     $v \leftarrow -\infty$ 
    for  $a, s$  in Successors(state) do  $v \leftarrow \text{Max}(v, \text{Min – Value}(s, \alpha, \beta))$ 
    if  $v \geq \beta$  then return  $v$ 
     $\alpha \leftarrow \text{Max}(\alpha, v)$ 
    return  $v$ 

function Min – Value(state,  $\alpha$ ,  $\beta$ )
    if Terminal – Test(state) then return Utility(state)
     $v \leftarrow +\infty$ 
    for  $a, s$  in Successors(state) do  $v \leftarrow \text{Min}(v, \text{Max – Value}(s, \alpha, \beta))$ 
    if  $v \leq \alpha$  then return  $v$ 
     $\beta \leftarrow \text{Min}(\beta, v)$ 
    return  $v$ 

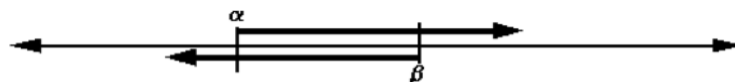
```

the maximum lower bound of possible solutions β Beta is the minimum upper bound of possible solutions, Thus, when any new node is being considered as a possible path to the solution, it can only work if: $\alpha \leq N \leq \beta$ where N is the current estimate of the value of the node.

To visualize this, we can use a number line. At any point in time, alpha and beta are lower and upper bounds on the set of possible solution values, like so: As the problem progresses, we can assume restrictions about the range of

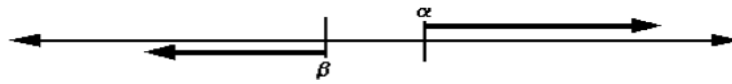


possible solutions based on min nodes (which may place an upper bound) and max nodes (which may place a lower bound). As we move through the search tree, these bounds typically get closer and closer together: This convergence



is not a problem as long as there is some overlap in the ranges of alpha and beta. At some point in evaluating a node, we may find that it has moved one of the bounds such that there is no longer any overlap between the ranges of

alpha and beta: At this point, we know that this node could never result in a



solution path that we will consider, so we may stop processing this node. In other words, we stop generating its children and move back to its parent node. For the value of this node, we should pass to the parent the value we changed which exceeded the other bound.

Example :

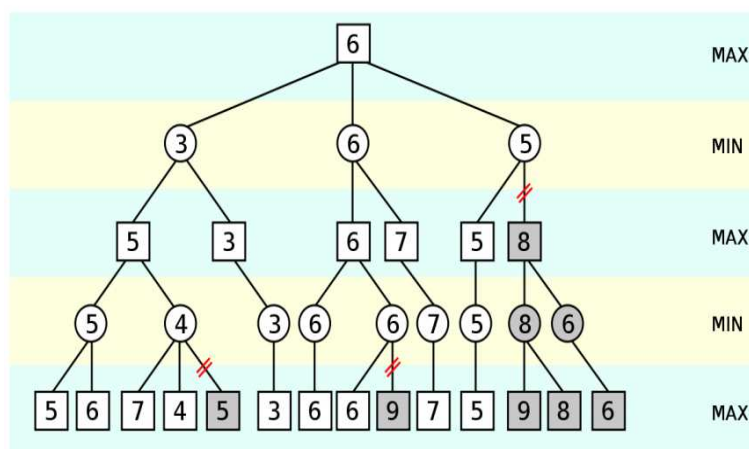


Figure 23: MinMax with Alpha Beta Algorithm

4.6 Summary

We discussed making game cautious and intelligent by implementing Minimax with the Alpha beta pruning algorithm in this section. Next section will explain development of chess game.

5 Development

5.1 Introduction

In this section, a more detailed explanation of the design of our project will be given.

5.2 Design

A game of chess involves chess pieces, and a chessboard. The chessboard is an 8 by 8 grid. The initial configuration of the pieces is as follows:



Figure 24: Chess Board with pieces

We proposed a design in which each type of chess pieces is represented by a separate class. We planned to implement 9 different classes for that purpose as follows: king, queen, knight, bishop, rook, pawn, Interface and Algorithm.

5.3 Classes

Here, we will summarize the general details of the classes in our object-oriented program.

5.3.1 MainClass.java Class

This is the main class of the project which makes object from all existing classes and runs the necessary functions of the classes.

5.3.2 King.java Class

This is the class of depend to the king movement and its rules in chess, that is to say where can the king move in chess board? What is the valid moves for the king? How it wakes the move? This class explain all this issues.

5.3.3 Queen.java Class

This is the class of depend to the queen movement and its rules in chess, that is to say where can the queen move in chess board? What is the valid moves for the queen? How it wakes the move? This class explain all this issues.

5.3.4 Knight.java Class

This is the class of depend to the knight movement and its rules in chess, that is to say where can the knight move in chess board? What is the valid moves for the knight? How it wakes the move? This class explain all this issues.

5.3.5 Bishop.java Class

This is the class of depend to the bishop movement and its rules in chess, that is to say where can the bishop move in chess board? What is the valid moves for the bishop? How it wakes the move? This class explain all this issues.

5.3.6 Rook.java Class

This is the class of depend to the rook movement and its rules in chess, that is to say where can the rook move in chess board? What is the valid moves for the rook? How it wakes the move? This class explain all this issues.

5.3.7 Pawn.java Class

This is the class of depend to the pawn movement and its rules in chess, that is to say where can the pawn move in chess board? What is the valid moves for the pawn? How it wakes the move? This class explain all this issues.

5.3.8 Algorithm.java Class

This is the class which make the chess perception and intelligent, this class consist in the Minimax with the alpha beta pruning algorithm and evaluation

function.

5.3.9 InterFace.java Class

As we design chess graphically, thus we created this class for graphically user functions.

5.4 Outline of the Work Done

Whereas the ultimate goal of project is to design the chess computerized and intelligent, for achieving this goal, first of all we developed the game for two players to play the game according to rules of the chess as real as possible. For this seven classes has been create in java computer programing language, they consist in: MainClass.java, King.java, Queen.java, Knight.java, Rook.java, Bishop.java, Pawn.java.

The ultimate function of the MainClass.java consist in: creating chess board and its elements: producing pieces movement on the board and implement the whole rules of the chess by creating object from existing six classes (King.java, Queen.java, Knight.java, Rook.java, Bishop.java, and Pawn.java).figure 25 illustrate Classes in Chess game

After retrieving the outcome and the result of the project. We have successfully achieved to create a graphically environment for user, for this we create a class name InterFace.java.

Finally, we create a class named Algorithm.java which includes the whole intelligent functions that has been used for the intelligence in the game.

5.5 Model

We designed chess game fully in graphically environment, that makes the users to make the move relax and ease.figure 26 illustrate Environment Chess Game. For doing the move user recommended to select the piece on the chess board by pressing and keeping the mouse key and drug or transform the selected piece to the specific and valid cell of the move.

If a player play against computer after a move a little delay will be generate by computer to search the valid and optimal move. figure 28 illustrate computer move.

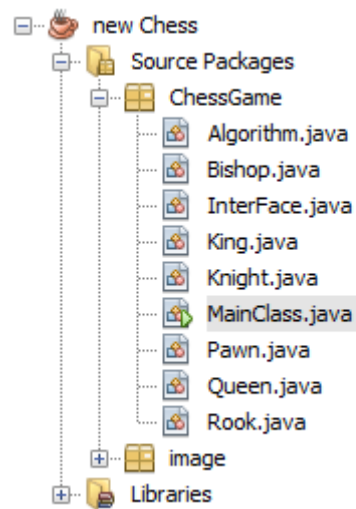


Figure 25: Classes in Chess Game

Assume that two player play the chess, if the first player does his/her move the turn will automatically change from player first to the second one, and will not let to the first player to do the move, that is to say the second player must made a move than the first and then the second and so on..figure 27 illustrate Human move.

5.6 Summary

The methodology of chess implementation has discussed in this section, the next section will explore more details.



Figure 26: Environment Chess Game



Figure 27: Human Move

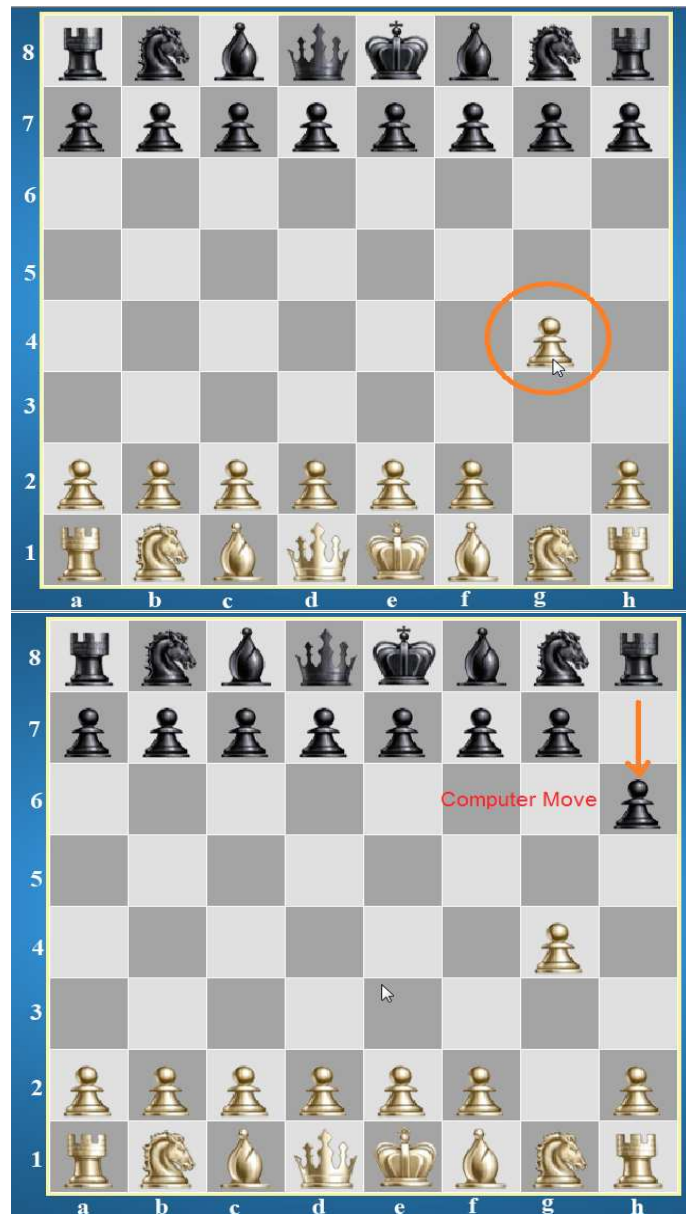


Figure 28: Computer Move

6 Implementation

6.1 Introduction

This section explores fully chess implementation details, we tried to explain the whole implementation classes and their methods.

As we mentioned in previous section intelligent chess in graphically user interface needed a computer programming language for implementation, therefore we have selected java which is full object oriented and runs in any platform that will lead the chess to run in any platform form.

After choosing the programming language for making the program and the codes manageable and flexible we need to define and create the classes first of all we computerize the chess, second we are going to create the graphically user interface and finally the intelligence to the game. The classes are: MainClass.java, King.java, Queen.java, Knight.java, Rook.java, Bishop.java, Pawn.java, interface.java, Algorithm.java.

We gave some information in the last section about these classes, this section will focus with more details especially the function and methods of the classes.

6.2 MainClass.java Class

This is the first class of the project which consists in:

- **main(String[] args)** Is the first function of the main class, that makes the bases of the program, in other words this is the starting function of the project.
- **possibleMovesW()** This function is for returning the valid moves of white pieces.
- **PossibleMovesB()** This is the same above functions and generating the valid moves for Black pieces.
- **makeMoveB(String move)** This is function is for doing a move for Black Pieces.
- **makeMoveW(String move)** this is the same as the last function it does a move for White Piece.

- **undoMoveB(String move)** this function doing the undo action for Black pieces.
- **undoMoveW(String move)** this function doing the same thing as the last function for white pieces.

6.3 King.java Class

This class include the whole valid rules for the piece of king in the game.

- **possibleKW(int i)** this function return the all valid moves for white pieces.
- **possibleKB(int i)** this function return the all valid moves for black pieces.
- **kingSafeW()** this function is for checking the state of the white king.
- **kingSafeB()** this function is for checking the state of the black king.

6.4 Queen.java Class

This class includes the entire functions depend to the Queen in the project.

- **possibleQW(int i)** this function return all the valid moves for the queen.
- **possibleQB(int i)** this function return all the valid moves for the queen.

6.5 Knight.java Class

This class includes the whole rules concerning to the knight.

- **possibleNW(int i)** this function return all possible and valid moves of the whit knight.
- **possibleNB(int i)** this function return all possible and valid moves of the black knight.

6.6 Bishop.java Class

This class consist the all rules of the Bishop.

- **possibleBW(int i)** this function return the all valid moves of the white Bishop.

- **possibleBB(int i)** this function return the all valid moves of the black Bishop.

6.7 Rook.java Class

This class explore the whole rules related to Rook.

- **possibleRW(int i)** the white rooks all valid move returns by this function.
- **possibleRB(int i)** this is the same function for the black.

6.8 Pawn.java Class

This class include the all rules related to the pawn.

- **possiblePW(int i)** this function returns the whole valid moves for the white pawn.
- **possiblePB(int i)** this function returns the whole valid moves for the black pawn.

6.9 InterFace.java class

This class consists the whole materials concerning to the graphical environment of the project.

- **paintComponent(Graphics g)** this function generate the chess board and its elements for graphically user interface.

6.10 Algorithm.java Class

This class includes the all functions related to the intelligency of the chess game.

- **alphaBeta(int depth,int beta,int aplha,String move,in player)** this function implement the alpha beta pruning algorithm.
- **rateAttack()** this function is for evaluation and identification of risks.
- **rateMaterial()** this function for identification of the remaining pieces on the board.

- **rateMoveablity(int listLength,int depth ,int material)** this function is for identifying all the move related to the computer game.
- **ratePositional(int material)** this is for finding the location of computer movement pieces.
- **rating(int list,int depth)** This function rates the whole piece in each possible moves of computer.

7 Conclusion

7.1 Conclusion and future enhancement

The main focus of this chapter is on the work has done and future enhancement of the project, the first section discuss the conclusion and summary of the whole Final Project and the second section is dedicated to the future enhancement of the project. Conclusion This Final Project aimed to create a practical and useable product, which can be considered as software and even an educational tool. A real life situation, which is a chess game in the current case, was modeled without any restrictions and based on its modelers personal understanding. Another aim of this Final Project was to analyze two agent base approaches. It was a fascinating and quite helpful experience for us to observe the differences between the two approaches by producing a practical work rather than conducting just a theoretical research. The users will observe and realize the fact that it is a very open-ended model, allowing users to interact with it using their own imagination, which was the main idea behind all this effort. Hopefully, however, it has managed to lay the groundwork for further study of narrative in video games.

7.2 Future Enhancement

In this section suggestions for the future research are given. These suggestions can be categorized into seven categories.

1. Evaluation function: Additional work can be performed on the evaluation function to increase the playing strength. Although the evaluation function used in the program has more than 10 features, even more features can be implemented. For regular Chess there are many features that can be adapted to multi-player Chess to increase the playing strength. Especially, features representing endgame knowledge can be beneficial because no such feature is included in the current evaluation function. Additionally, experts for multi-player Chess can be searched to retrieve domain knowledge that can be used to add some more features to the evaluation function. Currently, the evaluation function uses two features for the opening. The performance in this part of the game can be increased by building and using opening books. The weights of the different features

are tuned by hand or by just a few experiments. Therefore the playing strength can be increased by using machine learning techniques to tune the existing parameters of the evaluation function

2. Forward pruning: Forward pruning techniques are not used in the program. In quite some regular Chess programs, forward pruning techniques are applied. Therefore, it can be investigated whether applying forward pruning techniques like Null Move (Donninger, 1993), ProbCut (Buro, 1995) and MultiCut (Bjornsson and Marsland, 1999) can increase the playing strength.
3. Faster framework: As part of this document, a framework was implemented from scratch. This was the first time that the author of this document implemented a framework for playing a multi-player game. A few weeks were spent to optimize the framework, but nevertheless the framework can be even more efficient by changing some implementation details like using bitboards (Heinz, 1997; Hyatt, 1999; Reul, 2009) to represent the board.
4. Static move ordering: As explained in Subsection 4.3.5, the static move ordering is quite important for the performance of the proposed algorithms. The static move ordering in the program is quite simple. Spending more time to enhance the static move ordering could even increase the performance of BRS1,C1. One improvement can be to consider check moves. This is not considered in the program because it cost some computation time to identify check moves. But for the static move ordering for the non-searching opponents, this can be a promising approach to improve the quality of the move ordering. Another approach to improve the static move ordering can be the use of piece-square tables. Piece-square tables can be used to evaluate the position of the piece to move with respect to the kind of the moved piece.
5. Allow non-searching opponents to investigate more than one: move In the proposed algorithm BRS1,C1 all but one opponents play the best move regarding the static move ordering. When check moves are also considered in the static move ordering as proposed above, the question arises whether it is better to expect the opponent to play a good capture move or a check move. It could be a promising idea to propose an

algorithm where the non-searching opponents are allowed to perform a search with a small branching factor like 2 or 3. In such an algorithm it would be possible to consider check moves and also capture moves for the non-searching opponents. This modification would increase the complexity of the tree, but it would still be much smaller than a paranoid-tree. It has to be tested whether the better estimation of the opponents moves is worth the higher complexity of the tree.

6. Other domains: The proposed algorithms are only tested in the domain of multi-player Chess. It has to be tested whether the best of the proposed algorithms, BRS1,C1, is also able to outperform the existing algorithms in other domains. The algorithm can be tested in other capture games and also in non-capture games. Additionally, it can be tested in games with imperfect information. It is from special interest to see how the properties of the different games have an influence on the performance of BRS1,C1. In multi-player Chess, the evaluation function is more time consuming than the move generation. In other games where the move generation is more complex or the board position is quite easy to evaluate, the larger number of internal nodes compared to best-reply can be such a disadvantage that it is not worth to let the non-searching opponents play a move. Moreover, in multi-player Chess the opponents moves have a quite large influence on the position of the root player. Thus, it is quite important to let each opponent play a move. For other games where the opponents moves does not have such a large influence on the root players position, for instance racing-games, it has to be tested whether BRS1,C1 is also able to outperform best-reply.

Bibliography

References

- [1] *Artificial Intelligence (FORSYTH PONCE Computer Vision: A Modern Approach GRAHAM ANSI Common Lisp JURAFSKY MARTIN speech and Language; Processing NEAPOLITAN Learning Bayesian Networks RUSSELL NORVIG Artificial Intelligence: A Modern Approach)*
- [2] *COMPUTER CHESS: EXPLORING SPEED AND INTELLIGENCE (PAUL RAYMOND STEVENS A Final Project Submitted to The Honors College in Partial Fulfillment of the Bachelors degree With Honors in Computer Science THE UNIVERSITY OF ARIZONA May 2009).*
- [3] *DEVELOPMENT OF AN ADAPTIVE CHESS PROGRAM (Eda Okur Selin Kavuzlu Boazii University June 2011).*
- [4] *CSE 212 Final Project: Chess Game (Rebecca Oehmen; Cecilia Hopkins; Ryan Lichtenwalter).*
- [5] *Algorithms for solving sequential (zero-sum) games Main case in these slides: chess (Tuomas Sandholm).*
- [6] *Bertsekas, D. and Tsitsiklis, J. N. (2002). Introduction to Probability. Athena Scientific, Belmont, Massachusetts.*
- [7] *Cournot, A. (Ed.). (1838). Recherches sur les principes mathématiques de la théorie des richesses. L. Hachette, Paris.*
- [8] *Berliner, H. J. and Ebeling, C. (1989). Pattern knowledge and search: The SUPREM architecture. Artificial Intelligence.*
- [9] *, M. R. B. (Ed.). (1977). Advances in Computer Chess 1. Edinburgh University Press, Edinburgh, Scotland.*
- [10] *Brooks, R. A. (1990). Elephants don't play chess. Autonomous Robots.*
- [11] *Bernstein, A. and Roberts, M. (1958). Computer vs. chess player. Scientific American.*

- [12] *Beal, D. F. (1980). An analysis of minimax. In Clarke, M. R. B. (Ed.), Advances in Computer Chess.*

Web reference

<https://www.cs.tcd.ie/Glenn.Strong/3d5/minimax-notes>
<http://www.cs.swarthmore.edu/meeden/Minimax/Minimax.html>
<http://www.chess.com/learn-how-to-play-chess>
<http://www.code2learn.com/2012/01/minimax-algorithm-tutorial.html>
http://www.gamedev.net/page/resources/_/technical/artificial-intelligence/chess-programming-part-i-getting-started-r1014
http://www.gamedev.net/page/resources/_/technical/artificial-intelligence/chess-programming-part-ii-data-structures-r1046
http://www.gamedev.net/page/resources/_/technical/artificial-intelligence/chess-programming-part-iii-move-generation-r1126
http://www.gamedev.net/page/resources/_/technical/artificial-intelligence/chess-programming-part-iv-basic-search-r1171
http://www.gamedev.net/page/resources/_/technical/artificial-intelligence/chess-programming-part-vi-evaluation-functions-r1208
http://www.gamedev.net/page/resources/_/technical/artificial-intelligence/chess-programming-part-v-advanced-search-r1197
<https://www.youtube.com/watch?v=a-2uSg4Kvb0list=PLQV5mozTHmaffB0rBsD6m9VN1azgo5wXl>
<https://www.youtube.com/watch?v=jwRRJmsWFn4list=PLQV5mozTHmaffB0rBsD6m9VN1azgo5wXlindex=2>
<https://www.youtube.com/watch?v=NMAdwramt1glist=PLQV5mozTHmaffB0rBsD6m9VN1azgo5wXlindex=3>
<https://www.youtube.com/watch?v=1PzDYwDsXzslist=PLQV5mozTHmaffB0rBsD6m9VN1azgo5wXlindex=4>
<https://www.youtube.com/watch?v=LNEGkBxlpSulist=PLQV5mozTHmaffB0rBsD6m9VN1azgo5wXlindex=5>
<https://www.youtube.com/watch?v=HkuqmWzE08Alist=PLQV5mozTHmaffB0rBsD6m9VN1azgo5wXlindex=6>
<https://www.youtube.com/watch?v=ShenZF9Kmykindex=7list=PLQV5mozTHmaffB0rBsD6m9VN1azgo5wXl>
<https://www.youtube.com/watch?v=6jPE0aAWsvYlist=PLQV5mozTHmaffB0rBsD6m9VN1azgo5wXlindex=8>

<https://www.youtube.com/watch?v=Fy3A_BsBktUindex = 9list = PLQV5mozTHmaffB0rBsD6m9VN1azgo5wXl>
<https://www.youtube.com/watch?v=46V7mBRVwXYlist=PLQV5mozTHmaffB0rBsD6m9VN1azgo5wXlindex=10>
<https://www.youtube.com/watch?v=kWLO7HG9M_Ilist = PLQV5mozTHmaffB0rBsD6m9VN1azgo5wXlindex=11>
<https://www.youtube.com/watch?v=cZOdExE1NaUindex=12list=PLQV5mozTHmaffB0rBsD6m9VN1azgo5wXl>
<https://www.youtube.com/watch?v=P-qGwTNBwdQindex=13list=PLQV5mozTHmaffB0rBsD6m9VN1azgo5wXl>
<https://www.youtube.com/watch?v=d_K0Jjy9BXMindex = 14list = PLQV5mozTHmaffB0rBsD6m9VN1azgo5wXl>
<https://www.youtube.com/watch?v=ks4MaF90Jaklist=PLQV5mozTHmaffB0rBsD6m9VN1azgo5wXlindex=15>
<https://www.youtube.com/watch?v=h4nHxLC8pp4index=16list=PLQV5mozTHmaffB0rBsD6m9VN1azgo5wXl>
<https://www.youtube.com/watch?v=ngO6j5FYdG8index=17list=PLQV5mozTHmaffB0rBsD6m9VN1azgo5wXl>
<https://www.youtube.com/watch?v=fJ4uQpkn9V0list=PLQV5mozTHmaffB0rBsD6m9VN1azgo5wXlindex=18>
<https://www.youtube.com/watch?v=UZLnDvdeNo8list=PLQV5mozTHmaffB0rBsD6m9VN1azgo5wXlindex=19>
<https://www.youtube.com/watch?v=Wyh-5P5-7U8list=PLQV5mozTHmaffB0rBsD6m9VN1azgo5wXlindex=20>
<https://www.youtube.com/watch?v=8xBjxYHVwxMindex=21list=PLQV5mozTHmaffB0rBsD6m9VN1azgo5wXl>
<https://www.youtube.com/watch?v=tjsciMjqixAlist=PLQV5mozTHmaffB0rBsD6m9VN1azgo5wXlindex=22>
<https://www.youtube.com/watch?v=1iisYlFdh6sindex=23list=PLQV5mozTHmaffB0rBsD6m9VN1azgo5wXl>
<https://www.youtube.com/watch?v=s6wqQV8s96slist=PLQV5mozTHmaffB0rBsD6m9VN1azgo5wXlindex=24>
<https://www.youtube.com/watch?v=3SiyqU7JXeUlist=PLQV5mozTHmaffB0rBsD6m9VN1azgo5wXlindex=25>
<https://www.youtube.com/watch?v=cDz2u8tawQslist=PLQV5mozTHmaffB0rBs>

D6m9VN1azgo5wXlindex=26

https://www.youtube.com/watch?v=akfcmo_sZvo $index = 27$ $list = PLQV5moz$

THmaffB0rBsD6m9VN1azgo5wXl

<https://www.youtube.com/watch?v=fyD1Fa7-y-Y> $index = 28$ $list = PLQV5moz$

THmaffB0rBsD6m9VN1azgo5wXl

<https://www.youtube.com/watch?v=w-7GKtTsG3Y> $index = 29$ $list = PLQV5moz$

THmaffB0rBsD6m9VN1azgo5wXl

<https://www.youtube.com/watch?v=I1J1UJ3yMXQ> $index = 30$ $list = PLQV5moz$

THmaffB0rBsD6m9VN1azgo5wXl

https://www.youtube.com/watch?v=dqL0L6tNz_M $list = PLQV5mozTHmaffB0rBs$

D6m9VN1azgo5wXlindex=31

Appendix A: Source Codes

MainClass.java class

```
package ChessGame;

import static ChessGame.Interface.window;
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.TextArea;
import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Stack;
import javax.swing.Box;
import javax.swing.ButtonGroup;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JRadioButton;
import javax.swing.KeyStroke;
import javax.swing.border.LineBorder;
import javax.swing.border.TitledBorder;

*
* @author abhar
*

static String chessBoard[] [] = {
    {"d", "n", "b", "q", "k", "b", "n", "e"},
```

```

{"s", "t", "u", "v", "w", "x", "y", "z"},
{"", "", "", "", "", "", "", ""},
{"", "", "", "", "", "", "", ""},
{"", "", "", "", "", "", "", ""},
{"", "", "", "", "", "", "", ""},
{"", "", "", "", "", "", "", ""},
{"S", "T", "U", "V", "W", "X", "Y", "Z"},
{"D", "N", "B", "Q", "K", "B", "N", "E"}
};

static int countRowW = 0, countColW = 0, countRowB = 0, countColB
= 1;

static int kingPositionB, kingPositionW;
kingPositionB = 0;
kingPositionW = 0;
while ("K" . equals (chessBoard[kingPositionW
kingPositionW++;
}
while ("k" . equals (chessBoard[kingPositionB
kingPositionB++;
}
jfarme n1 = new jfarme();
System.out.println(" : white");
}
public static String possibleMovesW() {
String list = "";
for (inti = 0; i < 64; i++) {
if (Character . isUpperCase (chessBoard[i
switch (chessBoard[i
case "S":
list += Pawn.possiblePSW (i);
break;
case "T":
list += Pawn.possiblePTW (i);
break;
case "U":
list += Pawn.possiblePUW (i);

```



```

break;
case "V":
break;
}
}
}
return list.replaceAll(" . . . k", "");
}
public static String posibleMovesB() {
String list = "";
for (inti = 63; i > -1; i-- = 1) {
if (Character.isLowerCase(chessBoard[i
switch (chessBoard[i
case "s":
list += Pawn.posiblePSB(i);
break;
case "t":
list += Pawn.posiblePTB(i);
list += Pawn.posiblePZB(i);
break;
case "d":
list += Rook.posibleRDB(i);
break;
case "e":
list += Rook.posibleREB(i);
break;
case "n":
list += Knight.posibleNB(i);
break;
case "b":
list += Bishop.posibleBB(i);
break;
case "q":
list += Queen.posibleQB(i);
break;

```

```

case "k":
list += King.possibleKB(i);
break;
}
}
}
return list.replaceAll("...K", "");
}
public static void makeMoveW(String move) {
if (move.charAt(4) != 'C' & move.charAt(4) != 'P' & move.charAt(4)
= 'T' & move.charAt(4) != 'S' & move.charAt(4)
)= 'U' & move.charAt(4) != 'V' & move.charAt(4) != 'W' & move.charAt(4)
= 'X' & move.charAt(4) != 'Y' & move.charAt(4) != 'Z' {
if (choseS == false) {
captuerW(move);
undoMove.push(move);
}
cou = 1;
chessBoard[Character.getNumericValue(move.charAt(2))] [
Character.getNumericValue(move.charAt(3))] = chessBoard [
Character.getNumericValue(move.charAt(0))] [
Character.getNumericValue(move.charAt(1))];
chessBoard[Character.getNumericValue(move.charAt(0))] [
Character.getNumericValue(move.charAt(1))] = "";
if ("K" . equals (chessBoard [Character.getNumericValue (
move.charAt(2))] [
Character.getNumericValue (move.charAt(3))
) {
kingPositionW = 8 * Character.getNumericValue (move.charAt(2))
+ Character.getNumericValue (
move.charAt(3));
}
if (chessBoard [Character.getNumericValue (move.charAt(2))] [
Character.getNumericValue (move.charAt(3))] . equals (
chessBoard [Character.getNumericValue (move.charAt(0)) - 1]

```

```

left[Character·getNumericValue (
    move·charAt (1)) + Pawn·jState) {Stringarray [] =
    {"S", "T", "U", "V", "X", "Y", "Z", "W"};
    for (inti = 0; i < array·length; i++) {
        String P = array[i];
        if (P·equals ((chessBoard [Character·getNumericValue (
left(move·charAt (2)) [
    Character·getNumericValue (move·charAt (3))])
    & Pawn·statepT == false {
        String pawnP = chessBoard [Character·getNumericValue (
        move·charAt (0)) [
        Character·getNumericValue (move·charAt (1)) + Pawn·jState];
        countColW = 1;
        countRowW = 0;
        } else {
        countRowW++;
        countColW = 1;
        countRowW = 0;
        } else {
        countRowW++;
        }
        }
        chessBoard [Character·getNumericValue (move·charAt (0))] [
        Character·getNumericValue (move·charAt (1)) + Pawn·jState] = "";
        } else if (P·equals ((chessBoard [Character·getNumericValue (
        move·charAt (2)) [Character·getNumericValue (
        move·charAt (3))]) & Pawn·statepV == false {
        String pawnP = chessBoard [Character·getNumericValue (
        move·charAt (0)) [
        Character·getNumericValue (move·charAt (1)) + Pawn·jState];
        if ("".equals (pawnP)) {
        caputerB[countRowW] [countColW] = pawnP;
        if (countRowW == 7) {

```

```

countColW = 1;
countRowW = 0;
} else {
countRowW++;
}
}
chessBoard[Character · getNumericValue (move · charAt (0))] [
Character · getNumericValue (move · charAt (1)) + Pawn · jState] = "";
} else if (P · equals ((chessBoard [Character · getNumericValue (
move · charAt (2)) [Character · getNumericValue (
move · charAt (3))]&Pawn · statepW == false {
String pawnP = chessBoard[Character · getNumericValue (
move · charAt (0)) [
Character · getNumericValue (move · charAt (1)) +
Pawn · jState;
if (" " · equals (pawnP)) {
caputerB[countRowW] [countColW] = pawnP;
if (countRowW == 7) {
countColW = 1;
countRowW = 0;
} else {
countRowW++;
}
}
chessBoard[Character · getNumericValue (move · charAt (0))] [
Character · getNumericValue (move · charAt (1)) + Pawn · jState] = "";
} else if (P · equals ((chessBoard [Character · getNumericValue (
move · charAt (2)) [Character · getNumericValue (
move · charAt (3))]&Pawn · statepX == false {
String pawnP = chessBoard[Character · getNumericValue (
move · charAt (0)) [
Character · getNumericValue (move · charAt (1)) +
Pawn · jState;
if (" " · equals (pawnP)) {
caputerB[countRowW] [countColW] = pawnP;

```

```

if (countRowW == 7) {
countColW = 1;
countRowW = 0;
} else {
countRowW++;
}
}
chessBoard[Character · getNumericValue (move · charAt (0))] [
Character · getNumericValue (move · charAt (1)) + Pawn · jState] = "";
} else if (P · equals ((chessBoard [Character · getNumericValue (
move · charAt (2)) [Character · getNumericValue (
move · charAt (3))]&Pawn · statepY == false {
String pawnP = chessBoard[Character · getNumericValue (
move · charAt (0)) [
Character · getNumericValue (move · charAt (1)) +
Pawn · jState;
if (" " · equals (pawnP)) {
caputerB[countRowW] [countColW] = pawnP;
if (countRowW == 7) {
countColW = 1;
countRowW = 0;
} else {
countRowW++;
}
}
}
chessBoard[Character · getNumericValue (move · charAt (0))] [Character · getNumericValue
(move · charAt (1)) + Pawn · jState] = "";
} else if (P · equals ((chessBoard [Character · getNumericValue (
move · charAt (2)) [Character · getNumericValue (
move · charAt (3))]&Pawn · statepZ == false {
String pawnP = chessBoard[Character · getNumericValue (move · charAt (0))]
[Character · getNumericValue (move · charAt (1)) +
Pawn · jState;
if (" " · equals (pawnP)) {
caputerB[countRowW] [countColW] = pawnP;

```

```

if (countRowW == 7) {
countColW = 1;
countRowW = 0;
} else {
countRowW++;
}
}
if (" " . equals (pawnP)) {
caputerB[countRowW][countColW] = pawnP;
if (countRowW == 7) {
countColW = 1;
countRowW = 0;
} else {
countRowW++;
}
}
chessBoard[1][Character . getNumericValue (move . charAt (0))] = "";
pState = false;
} else {
if (choseS == false) {chessBoard [7][Character . getNumericValue (move . charAt (3))] = "
chessBoard[7][Character . getNumericValue (move . charAt (3))] = "E";
kingPositionW = 56 + Character . getNumericValue (move . charAt (2));
}
}
public static void undoMoveW(Stringmove) {
cou = 0;
if (move . charAt (4) =' C' & move . charAt (4) =' P') {
chessBoard[Character . getNumericValue (move . charAt (0))][
Character . getNumericValue (move . charAt (1))] = chessBoard [
Character . getNumericValue (move . charAt (2))][
Character . getNumericValue (move . charAt (3))];
chessBoard[Character . getNumericValue (move . charAt (2))][
Character . getNumericValue (move . charAt (3))] = String.valueOf (move . charAt (4));
if (" K" . equals (chessBoard [Character . getNumericValue (move . charAt (0))][
Character . getNumericValue (move . charAt (1))]) {

```

```

kingPositionW = 8 * Character.getNumericValue (move · charAt (0)) +
Character.getNumericValue (move · charAt (1));
}
} else if (move · charAt (4) == ' P') {
chessBoard[1] [Character · getNumericValue (move · charAt (0))] = " S";
chessBoard[0] [Character · getNumericValue (move · charAt (1))] =
String.valueOf (move · charAt (2));
} else {
chessBoard[7] [Character · getNumericValue (move · charAt (0))] = " K";
chessBoard[7] [Character · getNumericValue (move · charAt (1))] = " D";
chessBoard[7] [Character · getNumericValue (move · charAt (1))] = " E";
chessBoard[7] [Character · getNumericValue (move · charAt (2))] = "";
chessBoard[7] [Character · getNumericValue (move · charAt (3))] = "";
kingPositionW = 56 + Character.getNumericValue (move · charAt (0));
}
}
public static void makeMoveB(String move) {
if (move · charAt (4) == ' c' & move · charAt (4) == ' p' & move · charAt (4) == ' t'
& move · charAt (4) == ' s' & move · charAt (4) == ' u' & move · charAt (4) == ' v'
& move · charAt (4) == ' w' & move · charAt (4) == ' x' & move · charAt (4) == ' y'
& move · charAt (4) == ' z' {
if (MainClass · choseS == true) {
if (" s" · equals (chessBoard [Character · getNumericValue (move · charAt
(0))] [Character · getNumericValue (move · charAt (1))])) {
if (Math · abs (Character · getNumericValue (move · charAt (0)) –
Character.getNumericValue (move · charAt (2))) == 2
& Math · abs (Character · getNumericValue (move · charAt (1))
- Character.getNumericValue (move · charAt (3))) == 0 {
Pawn · statepS = false;
}
} else {
Pawn · statepS = true;
}
if (" t" · equals (chessBoard [Character · getNumericValue (move · charAt (
0))] [Character · getNumericValue (move · charAt (1))])) {

```

```

if (Math · abs (Character · getNumericValue (move · charAt (0))
- Character · getNumericValue (move · charAt (2))) == 2
& Math · abs (Character · getNumericValue (
move · charAt (1)) - Character · getNumericValue
(move · charAt (3))) == 0
{
Pawn · statepT = false;
}
} else {
Pawn · statepT = true;
}
if ("u" · equals (chessBoard [Character · getNumericValue (
move · charAt (0)) [
Character · getNumericValue (move · charAt (1))]) {if (Math · abs (
Character · getNumericValue (move · charAt (0)) - Character ·
getNumericValue (move · charAt (2))) == 2
& Math · abs (Character · getNumericValue (move · charAt (1))
- Character · getNumericValue (
move · charAt (3))) == 0
{
Pawn · statepU = false;
}
} else {
Pawn · statepU = true;
}
if ("v" · equals (chessBoard [Character · getNumericValue (move · charAt (0))]) [
Character · getNumericValue (move · charAt (1))]) {
if (Math · abs (Character · getNumericValue (move · charAt (0)) -
Character · getNumericValue (
move · charAt (2))) == 2
& Math · abs (Character · getNumericValue (move · charAt (1)) -
Character · getNumericValue (
move · charAt (3))) == 0
{
Pawn · statepV = false;

```



```

if ("x" · equals (chessBoard [Character · getNumericValue (Pawn · statepX = true;
}
if ("y" · equals (chessBoard [Character · getNumericValue (move · charAt (0))] [
Character · getNumericValue (move · charAt (1))]) {
if (Math · abs (Character · getNumericValue (move · charAt (0))
- Character · getNumericValue (move · charAt (2))) == 2
& Math · abs (Character · getNumericValue (move · charAt (1))
- Character · getNumericValue (move · charAt (3))) == 0 {
Pawn · statepY = false;
}
} else {
Pawn · statepY = true;
}
if ("z" · equals (chessBoard [Character · getNumericValue (move · charAt (0))] [
Character · getNumericValue (move · charAt (1))]) {
if (Math · abs (Character · getNumericValue (move · charAt (0))
- Character · getNumericValue (move · charAt (2))) == 2
& Math · abs (Character · getNumericValue (move · charAt (1))
- Character · getNumericValue (move · charAt (3))) == 0 {
Pawn · statepZ = false;
}
} else {
Pawn · statepZ = true;
}
}
if (choseS == false) {
captuerB(move);
cou = 0;
undoMove · push (move);
}
chessBoard [Character · getNumericValue (move · charAt (2))] [
Character · getNumericValue (move · charAt (3))] = chessBoard [Character · getNumericValue
(move · charAt (0))] [Character · getNumericValue
(move · charAt (1));
chessBoard [Character · getNumericValue (move · charAt (0))] [

```

```

Character·getNumericValue (move · charAt (1))] = "";
if ("k" · equals (chessBoard [Character · getNumericValue (move · charAt
(2)) [Character · getNumericValue (move · charAt (3))])) {
kingPositionB = 8 * Character·getNumericValue (move · charAt (2))+
Character·getNumericValue (move · charAt (3)); }
String array[] = {"s", "t", "u", "v", "x", "y", "z", "w"};
if (chessBoard [Character · getNumericValue (
move·charAt (2)) [Character · getNumericValue
(move · charAt (3))] · equals (chessBoard [
Character·getNumericValue (move · charAt (0)) + 1] [
Character·getNumericValue (move · charAt (1)) - Pawn · jState]) {
for (inti = 0; i < 8; i++) {
String p = array[i];
if (p · equals ((chessBoard [Character · getNumericValue (move · charAt
(2)) [Character · getNumericValue (move · charAt (3))]))&
Pawn.statePT == false {
String pawnP = chessBoard [Character · getNumericValue (move · charAt
(0)) [Character · getNumericValue (move · charAt (1)) - Pawn · jState];
if (" " · equals (pawnP)) {
caputerW[countRowB] [countColB] = pawnP;
if (countRowB == 7) {
countColB = 0;
countRowB = 0;
} else {
countRowW++;
}
}
}
cState = false;
chessBoard [Character · getNumericValue (move · charAt (0))] [
Character·getNumericValue (move · charAt (1)) - Pawn · jState] = "";
} else if (p · equals ((chessBoard [Character · getNumericValue
(move · charAt (2)) [Character · getNumericValue (move · charAt (3))]))
& Pawn.statePS == false {
String pawnP = chessBoard [Character · getNumericValue (move · charAt
(0)) [Character · getNumericValue (move · charAt (1)) - Pawn · jState];

```

```

if (" " · equals (caputerW [countRowB] [countColB])) {
caputerW[countRowB] [countColB] = pawnP;
if (countRowB == 7) {
countRowB = 0;
} else {
countRowW++;
}
}
cState = false;
chessBoard[Character · getNumericValue (move · charAt (0))] [
Character · getNumericValue (move · charAt (1)) - Pawn · jState] = "";
} else if (p · equals ((chessB
caputerW[countRowB] [countColB] = pawnP;
if (countRowB == 7) {
countColB = 0;
countRowB = 0;
} else {
countRowW++;
}
}
cState = false;
chessBoard[Character · getNumericValue (move · charAt (0))] [
Character · getNumericValue (move · charAt (1)) - Pawn · jState] = "";
} else if (p · equals ((chessBoard [Character · getNumericValue (
move · charAt (2)) [Character · getNumericValue (move · charAt (3))]))
}
cState = false;
chessBoard[Character · getNumericValue (move · charAt (0))] [
Character · getNumericValue (move · charAt (1)) - Pawn · jState] = "";
} else if (p · equals ((chessBoard [Character · getNumericValue
(move · charAt (2)) [Character · getNumericValue (move · charAt (3))]))
& Pawn · statePX == false {
String pawnP = chessBoard[Character · getNumericValue (move · charAt
(0)) [Character · getNumericValue (move · charAt (1)) - Pawn · jState];
if (" " · equals (pawnP)) {

```

```

caputerW[countRowB][countColB] = pawnP;
caputerW[countRowB][countColB] = pawnP;
if (countRowB == 7) {
countColB = 0;
countRowB = 0;
} else {
countRowW++;
}
}
cState = false;
chessBoard[Character · getNumericValue (move · charAt (0))][
Character · getNumericValue (move · charAt (1)) - Pawn · jState] = "";
} else if (p · equals ((chessBoard[Character · getNumericValue (
move · charAt (2))][Character · getNumericValue (move · charAt (3))]))
& Pawn · statePZ == false {
String pawnP = chessBoard[Character · getNumericValue (move · charAt (0))
[Character · getNumericValue (move · charAt (1)) - Pawn · jState];
if (" " · equals (pawnP)) {
caputerW[countRowB][countColB] = pawnP;
if (countRowB == 7) {
countColB = 0;
countRowB = 0;
} else {
countRowW++;
}
}
cState = false;
chessBoard[Character · getNumericValue (move · charAt (0))][
Character · getNumericValue (move · charAt (1)) - Pawn · jState] = "";
} else {
}
}
if (countRowB == 7) {
countColB = 0;
countRowB = 0;

```

```

} else {
countRowW++;
}
}
chessBoard[6][Character · getNumericValue (move · charAt (0))] = "";
chessBoard[7][Character · getNumericValue (move · charAt (1))]
= String · valueOf (move · charAt (3));
cState = true;
} else {
if (choseS == false) {
captuerB(move);
cou = 0;
undoMove · push (move);
}
chessBoard[0][Character · getNumericValue (move · charAt (0))] = "";
chessBoard[0][Character · getNumericValue (move · charAt (1))] = "";
chessBoard[0] l
System · exit (0);
window();
}
};
if (MainClass · posibleMovesW () · length () == 0) {
if (King · kingSafeW ()) {
JOptionPane · showOptionDialog (null, "GameEnd", "GameOver",
JOptionPane · DEFAULT_OPTION,
JOptionPane · PLAIN_MESSAGE, null, new Object []
{playagain, exitgame}, playagain);
} else {
JOptionPane · showOptionDialog (null, "BlackWON", "GameOver",
JOptionPane · DEFAULT_OPTION,
JOptionPane · PLAIN_MESSAGE, null, new Object []
{playagain, exitgame}, playagain);
}
}
}
}

```

```

public static void undoMoveB(Stringmove) {
    cou = 1;
    if (move · charAt (4) == ' c' & move · charAt (4) == ' p') {
        chessBoard[Character · getNumericValue (move · charAt (0))] [
            Character · getNumericValue (
                move · charAt (1)) = chessBoard [
                Character · getNumericValue (move · charAt (2))] [
                Character · getNumericValue (move · charAt (3))] ;
        chessBoard[Character · getNumericValue (move · charAt (2))] [
            Character · getNumericValue (
                move · charAt (3)) = String · valueOf (
                move · charAt (4));
        if ("k" · equals (chessBoard [Character · getNumericValue (
            move · charAt (0))] [
            Character · getNumericValue (move · charAt (1))
        ) {
            kingPositionB = 8 * Character · getNumericValue (move · charAt (0))
            + Character · getNumericValue (
                move · charAt (1));
        }
    } else if (move · charAt (4) == ' p') {
        chessBoard[6] [Character · getNumericValue (move · charAt (0))] = "";
        chessBoard[7] [Character · getNumericValue (move · charAt (1))] =
            String · valueOf (move · charAt (2));
    } else {
        chessBoard[0] [Character · getNumericValue (move · charAt (0))] = "k";
        chessBoard[0] [Character · getNumericValue (move · charAt (1))] = "d";
        chessBoard[0] [Character · getNumericValue (move · charAt (1))] = "e";
        chessBoard[0] [Character · getNumericValue (move · charAt (2))] = "";
        chessBoard[0] [Character · getNumericValue (move · charAt (3))] = "";
        kingPositionB = 0 + Character · getNumericValue (move · charAt (0));
    }
}

public static void makeMoveBP(Stringmove) {
    if (move · charAt (4) == ' p' || move · charAt (4) == ' t' || move · charAt (4)

```

```

    == 's' |||move · charAt(4) ==' u' |||move · charAt(4) ==' v' |||move ·
charAt(4) ==' w' |||move · charAt(4)
    == 'x' |||move · charAt(4) ==' y'
    |||move · charAt(4) ==' z' {
    if (choseS == false) {
    cou = 0;
    undoMove.push(move);
    }
    String pawnP = chessBoard[7][Character · getNumericValue(move · charAt(1))];
    if (" · equals(pawnP)) {
    caputerW[countRowB][countColB] = pawnP;
    if (countRowB == 7) {
    countColB = 0;
    countRowB = 0;
    } else {
    countRowW++;
    }
    }
    chessBoard[7][Character · getNumericValue(move · charAt(1))] =
String · valueOf(move · charAt(3));
    chessBoard[6][Character · getNumericValue(move · charAt(0))] = "";
    cState = false;
    }
    }
    public static class jfarme extends JFrame {
    jfarme() {
    super("");
    setLayout(new FlowLayout());
    JFrame f = new JFrame("ChESSGame");
    final jfarme thisFrame1 = this;
    JMenu fileMenu = new JMenu("Game");
    JMenu Help = new JMenu("Help");
    JMenuItem New1 = new JMenuItem("Newgameagainstcomputer");
    JMenuItem New = new JMenuItem("Newgameagainsthuman");
    JMenuItem undo = new JMenuItem("Undo");

```

```

JMenuItem change = new JMenuItem("Change Appearance");
New.setAccelerator(KeyStroke.getKeyStroke("F3"));
New1.setAccelerator(KeyStroke.getKeyStroke("F2"));
undo.setAccelerator(KeyStroke.getKeyStroke('Z', Toolkit.getDefaultToolkit().
getMenuShortcutKeyMask()));
New1.addActionListener(new ActionListener() {
@Override
public void actionPerformed(ActionEvent ae) {
if (in == 1) {
InterFace.back = "background1.jpg";
InterFace.bbak = "imf1.png";
InterFace.red = 224;
InterFace.green = 224;
InterFace.blue = 224;
back = "background1.jpg";
in = 0;
} else if (in == 0) {
InterFace.back = "background3.jpg";
InterFace.bbak = "imb3.png";
InterFace.red = 179;
InterFace.green = 204;
InterFace.blue = 218;
in = 1;
}
newGAME();
choseS = true;
}
};
New.addActionListener(new ActionListener() {
@Override
public void actionPerformed(ActionEvent ae) {
Algorithm.humanAsWhite = 1;
if (in == 1) {
InterFace.back = "background1.jpg";
InterFace.bbak = "imf1.png";

```



```

InterFace.red = 224;
InterFace.green = 224;
InterFace.blue = 224;
back = "background1.jpg";
in = 0;
} else if (in == 0) {
JMenuItem exit = new JMenuItem("Exit");
JMenuItem about = new JMenuItem("About");
JMenuItem help = new JMenuItem("Help");
final Icon iconn = new ImageIcon(getClass().getClassLoader().
getResource("image
about.addActionListener(new ActionListener() {
@Override
public void actionPerformed(ActionEvent) {
JOptionPane.showMessageDialog(null, "", "About",
JOptionPane.OK_OPTION, iconn;
}
});
help.addActionListener(new ActionListener() {
@Override
public void actionPerformed(ActionEvent) {
jhelpfarme jl = new jhelpfarme();
}
});
change.addActionListener(new ActionListener() {
@Override
public void actionPerformed(ActionEvent) {
setLayout(new FlowLayout());
JLabel lab = new JLabel("");
Icon pieceStyle1a = new ImageIcon(getClass().getClassLoader().
getResource("image
Icon pieceStyle1b = new ImageIcon(getClass().getClassLoader().
getResource("image
Icon pieceStyle2a = new ImageIcon(getClass().getClassLoader().
getResource("image

```

```

Icon pieceStyle2b = new ImageIcon(getClass().getClassLoader().
getResource("image
Icon blanka = new ImageIcon(getClass().getClassLoader().
getResource("image
Box box = Box.createHorizontalBox();
box.setBorder(newTitledBorder(newLineBorder(Color.DARK_GRAY),
" Select Piece Style ");
ButtonGroup group = new ButtonGroup();
JRadioButton blank = new JRadioButton("", blanka);
JRadioButton blank1 = new JRadioButton("", blanka);
JRadioButton BlackGold = newJRadioButton("", pieceStyle1a);
JRadioButton BlackWhite = newJRadioButton("", pieceStyle2a);
BlackGold.addActionListener(newActionListener() {
@Override
public void actionPerformed(ActionEvent) {
piece = "BlackGold.png";
}
});
BlackWhite.addActionListener(newActionListener() {
@Override
public void actionPerformed(ActionEvent) {
piece = "BlackWhite.png";
}
});
group.add(BlackGold);
group.add(BlackWhite);
BlackGold.setRolloverEnabled(true);
BlackGold.setRolloverIcon(pieceStyle1b);
BlackGold.setSelectedIcon(pieceStyle1b);
Box box2 = Box.createHorizontalBox();
box2.add(lab);
Box box3 = Box.createHorizontalBox();
box3.setBorder(newTitledBorder(newLineBorder
(Color.DARK_GRAY), "SelectBoard");
Icon board1 = new ImageIcon(getClass().getClassLoader().

```

```

getResource("image
Icon board2 = new ImageIcon(getClass().getClassLoader().
getResource("image
Icon board3 = new ImageIcon(getClass().getClassLoader().
boardS1.addActionListener(new ActionListener() {
@Override
public void actionPerformed(ActionEvent) {
back = "background1.jpg";
sq = "imf1.png";
red1 = 224;
green1 = 224;
blue1 = 244;
}
};
boardS2.addActionListener(new ActionListener() {
@Override
public void actionPerformed(ActionEvent) {
back = "background2.jpg";
sq = "a.png";
red1 = 120;
green1 = 190;
blue1 = 229;
}
};
boardS3.addActionListener(new ActionListener() {
@Override
public void actionPerformed(ActionEvent) {
back = "background3.jpg";
sq = "imb3.png";
red1 = 179;
green1 = 204;
blue1 = 218;
}
};
group1.add(boardS1);

```

```

group1.add (boardS2);
group1.add (boardS3);
box3.add (boardS1);
box3.add (boardS2);
box3.add (boardS3);
boardS1.setRolloverEnabled (true);
boardS1.setRolloverIcon (board1a);
boardS1.setSelectedIcon (board1a);
boardS2.setRolloverEnabled (true);
boardS2.setRolloverIcon (board2a);
boardS2.setSelectedIcon (board2a);
boardS3.setRolloverEnabled (true);
boardS3.setRolloverIcon (board3a);
boardS3.setSelectedIcon (board3a);
Box box4 = Box.createHorizontalBox ();
JButton button1 = new JButton("Ok");
JButton button2 = new JButton("Cancel");
button1.addActionListener (new ActionListener () {
@Override
public void actionPerformed(ActionEvent) {
Interface.pieceimage = piece;
Interface.back = back;
Interface.bbak = sq;
Interface.red = red1;
Interface.green
box4.add (button2);
Box box1 = Box.createVerticalBox ();
box1.add (box);
box1.add (box2);
box1.add (box3);
box1.add (box2);
box1.add (box4);
JOptionPane.showOptionDialog (thisFrame1, null, "Change Appearance",
JOptionPane.DEFAULT_OPTION, JOptionPane.PLAIN_MESSAGE,
null, new Object[] {box1}, null);

```

```

int orgH = (700 - frameH)
int orgW = (50 - frameW)
try {
f.setVisible(true);
f.setLocation(0,0);
InterFace ui = new InterFace();
f.add(bar, BorderLayout.NORTH);
f.add(ui, BorderLayout.CENTER);
int x = Toolkit.getDefaultToolkit().getScreenSize().width;
int y = Toolkit.getDefaultToolkit().getScreenSize().height - 37;
f.setSize(x,y);
f.setMinimumSize(new java.awt.Dimension(560,500));
f.setMaximumSize(new java.awt.Dimension(x,y));
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
} catch (Exception e) {
}
try {
for (javax.swing.UIManager.LookAndFeelInfo info : javax.swing.UIManager.
getInstalledLookAndFeels()) {
if ("Windows".equals(info.getName())) {
javax.swing.UIManager.setLookAndFeel(info.getClassName());
break;
}
}
} catch (ClassNotFoundException | InstantiationException |
IllegalAccessException — javax.swing.UnsupportedLookAndFeelException ex {
java.util.logging.Logger.getLogger(MainClass.class.getName
()) .log(java.util.logging.Level.SEVERE, null, ex); }
}
}

```

King.java Class

```

package ChessGame;

import static ChessGame.King.kingSafeW;
import static ChessGame.MainClass.chessBoard;
import static ChessGame.MainClass.kingPositionW;

```

```

import static ChessGame.MainClass.kingPositionB;
*
* @author a b h a r
*
public class King {
    static boolean stateRD = true, stateRE = true, stateRd = true, stateRe
= true;
    static boolean stateKW = false, stateKB = false;
    static boolean castleWhiteLong = true, castleWhiteShort = true, castle-
BlackLong = true,
    castleBlackShort = true;
    public static String posibleKW(int i) {
        String list = "", oldPiece;
        int r = i
        for (int j = 0; j < 9; j++) {
            if (j != 4) {
                try {
                    if (Character.isLowerCase(chessBoard[r - 1 + j
||||"] . equals(chessBoard[r - 1 + j
oldPiece = chessBoard[r - 1 + j
chessBoard[r][c] = "";
chessBoard[r - 1 + j
int kingTemp = kingPositionW;
kingPositionW = i + (j
if (kingSafeW()) {
    list = list + r + c + (r - 1 + j
    }
    chessBoard[r][c] = "K";
    chessBoard[r - 1 + j
    kingPositionW = kingTemp;
    }
    } catch (Exception e) {
    }
    }
    }
}

```

```

rookNewColumn,C
if ("K" . equals (chessBoard [7] [4]) &&stateRD&&castleWhiteLong&&"D" . equals (
chessBoard[7] [0] &&" " . equals
(chessBoard [7] [1]) &&" " . equals (chessBoard [7] [2]) &&" " . equals (chessBoard [7] [3]) {
boolean temp = false;
for (int j = 1; j <= 2; j++) {
MainClass.makeMoveW ("747" + j + "");
temp = (temp||!kingSafeW ());
MainClass.undoMoveW ("747" + j + "");
}
if (!temp) {
list += "4023C"; }
}
if ("K" . equals (chessBoard [7] [4]) &&stateRE&&castleWhiteShort&&"E" . equals (
chessBoard[7] [7] &&" " . equals
(chessBoard [7] [5]) &&" " . equals (chessBoard [7] [6]) {
boolean temp = false;
for (int j = 5; j <= 6; j++) {
MainClass.makeMoveW ("747" + j + "");
temp = (temp||!kingSafeW ());
MainClass.undoMoveW ("747" + j + "");
}
if (!temp) {
list += "4765C";
}
}
return list;
}
public static String posibleKB(int i) {
String list = "", oldPiece;
int r = i
for (int j = 0; j < 9; j++) {
if (j != 4) {
try {if (Character . isUpperCase (chessBoard [r - 1 + j
[c - 1 + j%3] . charAt (0)

```

```

|||||".equals(chessBoard
[r - 1 + j
oldPiece = chessBoard[r - 1 + j
chessBoard[r][c] = "";
chessBoard[r - 1 + j
int kingTemp = kingPositionB;
kingPositionB = i + (j
if (kingSafeB()) {
list = list + r + c + (r - 1 + j
}
chessBoard[r][c] = "k";
chessBoard[r - 1 + j
kingPositionB = kingTemp;
}
} catch (Exception) {
}
}
}
}
}
}
if ("k".equals(chessBoard[0][4]) &&stateRe&&castleBlackShort&&"e".equals(
chessBoard[0][7] &&"".equals
(chessBoard[0][5]) &&"".equals(chessBoard[0][6]) {
boolean temp = false;
for (int j = 5; j <= 6; j++) {
MainClass.makeMoveB("040" + j + "");
temp = (temp||!kingSafeB());
MainClass.undoMoveB("040" + j + "");
}
if (!temp) {
list += "4765c";
}
}
return list;
}

```



```

public static boolean kingSafeW() {
    int temp = 1;
    for (inti = -1; i <= 1; i++ = 2) {
        for (intj = -1; j <= 1; j++ = 2) {
            try {
                while (" " . equals (chessBoard [kingPositionW
                % 8 + temp * j]{
                    temp++;
                }
                if ("b" . equals (chessBoard [kingPositionW
                % 8 + temp * j
                |||"q" . equals (chessBoard [kingPositionW
                % 8 + temp * j]{
                    stateKW = true;
                    return false;
                }
            } catch (Exceptione) {
            }
            temp = 1;
        }
    }
    temp = 1;
    for (inti = -1; i <= 1; i++ = 2) {
        try {
            while (" " . equals (chessBoard [kingPositionW
            temp++;
        }
        if ("d" . equals (chessBoard [kingPositionW
        |||"e" . equals (chessBoard [kingPositionW
        |||"q" . equals (chessBoard [kingPositionW
        stateKW = true;
        return false;
    }
    } catch (Exceptione) {
    }
}

```

```

temp = 1;
try {
while (" " · equals (chessBoard [kingPositionW
temp++;
}
if ("d" · equals (chessBoard [kingPositionW
|||"e" · equals (chessBoard [kingPositionW
|||"q" · equals (chessBoard [kingPositionW
stateKW = true;
return false;
}
} catch (Exceptione) {
}
temp = 1;
}
for (inti = -1; i <= 1; i+ = 2) {
for (intj = -1; j <= 1; j+ = 2) {
try {
if ("n" · equals (chessBoard [kingPositionW
stateKW = true;
return false;
}
} catch (Exceptione) {
}
try {
if ("n" · equals (chessBoard [kingPositionW
stateKW = true;
return false;
}
} catch (Exceptione) {
}
}
}
}
} catch (Exceptione) {
}

```

```

}
try {
if ("s" . equals (chessBoard [kingPositionW
|||| "t" . equals (chessBoard [kingPositionW
|||| "u" . equals (chessBoard [kingPositionW
|||| "v" . equals (chessBoard [kingPositionW
|||| "w" . equals (chessBoard [kingPositionW
|||| "x" . equals (chessBoard [kingPositionW
|||| "y" . equals (chessBoard [kingPositionW
|||| "z" . equals (chessBoard [kingPositionW
stateKW = true;
return false;
}
} catch (Exception) {
}
}
for (inti = -1; i <= 1; i++) {
for (intj = -1; j <= 1; j++) {
if (i! = 0 || j! = 0) {
try {
if ("k" . equals (chessBoard [kingPositionW
stateKW = true;
return false;
}
} catch (Exception) {
}
}
}
}
stateKW = false;
return true;
}
public static boolean kingSafeB() {
int temp = 1;
for (inti = -1; i <= 1; i++ = 2) {

```

```

for (int j = -1; j <= 1; j++ = 2) {
try {
while (" " · equals (chessBoard [kingPositionB
temp++;
}
if ("B" · equals (chessBoard [kingPositionB
|||"Q" · equals (chessBoard [kingPositionB
stateKB = true;
return false;
}
} catch (Exception) {
}
temp = 1;
}
}
temp = 1;
for (int i = -1; i <= 1; i++ = 2) {
try {
while (" " · equals (chessBoard [kingPositionB
temp++;
}
if ("D" · equals (chessBoard [kingPositionB
|||"E" · equals (chessBoard [kingPositionB
|||"Q" · equals (chessBoard [kingPositionB
stateKB = true;
return false;
}
} catch (Exception) {
}
temp = 1; } catch (Exception) {
}
temp = 1;
}
for (int i = -1; i <= 1; i++ = 2) {
for (int j = -1; j <= 1; j++ = 2) {

```

```

try {
if ("N" · equals (chessBoard [kingPositionB
stateKB = true;
return false;
}
} catch (Exceptione) {
}
try {
if ("N" · equals (chessBoard [kingPositionB
stateKB = true;
return false;
}
} catch (Exceptione) {
}
}
}
if (kingPositionB <= 47) {
try {if ("S" · equals (chessBoard [kingPositionB
|||"T" · equals (chessBoard [kingPositionB
|||"U" · equals (chessBoard [kingPositionB
|||"V" · equals (chessBoard [kingPositionB
|||"W" · equals (chessBoard [kingPositionB
|||"X" · equals (chessBoard [kingPositionB
|||"Y" · equals (chessBoard [kingPositionB
|||"Z" · equals (chessBoard [kingPositionB
stateKB = true;
return false;
}
} catch (Exceptione) {
}
try {
if ("S" · equals (chessBoard [kingPositionB
|||"T" · equals (chessBoard [kingPositionB
|||"U" · equals (chessBoard [kingPositionB
|||"V" · equals (chessBoard [kingPositionB

```

```

|||| "W" . equals (chessBoard [kingPositionB
|||| "X" . equals (chessBoard [kingPositionB
|||| "Y" . equals (chessBoard [kingPositionB
|||| "X" . equals (chessBoard [kingPositionB
stateKB = true;
return false; }
} catch (Exception e) {
}
}
for (inti = -1; i <= 1; i++) {
for (intj = -1; j <= 1; j++) {
if (i != 0 || j != 0) {
try {
if ("K" . equals (chessBoard [kingPositionB
stateKB = true;
return false;
}
} catch (Exception e) {
}
}
}
}
stateKB = false;
return true;
}
}

```

Queen

```

package ChessGame;

import static ChessGame.MainClass.chessBoard;
*
* @author a b h a r
*

public class Queen {
public static String posibleQW(inti) {
String list = "", oldPiece;

```

```

int r = i
int temp = 1;
for (int j = -1; j <= 1; j++) {
for (int k = -1; k <= 1; k++) {
if (j = 0 || k = 0) {
try {
while ("" . equals (chessBoard[r + temp * j] [c + temp * k] {
oldPiece = chessBoard[r + temp * j][c + temp * k];
chessBoard[r][c] = "";
chessBoard[r + temp * j][c + temp * k] = "Q";
if (King . kingSafeW ()) {
list = list + r + c + (r + temp * j) + (c + temp * k) + oldPiece;
}
chessBoard[r][c] = "Q";
temp = 1;
}
}
}
return list;
}
public static String posibleQB(int i) {
String list = "", oldPiece;
int r = i
int temp = 1;
for (int j = -1; j <= 1; j++) {
for (int k = -1; k <= 1; k++) {
if (j = 0 || k = 0) {
try {
while ("" . equals (chessBoard[r - temp * j] [c - temp * k] {
oldPiece = chessBoard[r - temp * j][c - temp * k];
chessBoard[r][c] = "";
chessBoard[r - temp * j][c - temp * k] = "q";
if (King . kingSafeB ()) {
list = list + r + c + (r - temp * j) + (c - temp * k) + oldPiece;
}
}
}
}
}
}

```

```

chessBoard[r][c = "q";
chessBoard[r - temp * j][c - temp * k = oldPiece;
temp++;
}
if (Character.toUpperCase(chessBoard[r - temp * j]
    .charAt(0) && "K") .equals
(chessBoard[r - temp * j][c - temp * k] {
oldPiece = chessBoard[r - temp * j][c - temp * k;
}

```

knight.java class

```

package ChessGame;
import static ChessGame.MainClass.chessBoard;
*
* @author a b h a r
*
public class Knight {
public static String possibleNW(int i) {
String list = "", oldPiece;
int r = i
for (int j = -1; j <= 1; j++ = 2) {
for (int k = -1; k <= 1; k++ = 2) {
try {
if (Character.toLowerCase(chessBoard[r + j]
    .charAt(0) || "" .equals (
chessBoard[r + j][c + k * 2] {
oldPiece = chessBoard[r + j][c + k * 2;
chessBoard[r][c = "";
chessBoard[r + j][c + k * 2 = "N";
if (King.kingSafeW()) {
list = list + r + c + (r + j) + (c + k * 2) + oldPiece;
}
chessBoard[r][c = "N";
chessBoard[r + j][c + k * 2 = oldPiece;
}
} catch (Exception e) {

```



```

}
try {
if (Character · isLowerCase (chessBoard[r + j * 2]
· charAt (0) |||"" · equals (chessBoard
[c + k{oldPiece = chessBoard[r + j * 2] [c + k;
chessBoard[r[c = "";
chessBoard[r + j * 2][c + k = "N";
if (King · kingSafeW ()) {
list = list + r + c + (r + j * 2) + (c + k) + oldPiece; }
chessBoard[r[c = "N";
chessBoard[r + j * 2][c + k = oldPiece; }
} catch (Exception) {
}
}
}
return list;
}
public static String posibleNB(inti) {
String list = "", oldPiece;
int r = i
for (intj = -1; j <= 1; j + = 2) {
for (intk = -1; k <= 1; k + = 2) {
try {
if ((Character · isUpperCase (chessBoard[r - j]
· charAt (0) |||"" · equals (chessBoard[r - j]
&&"K" · equals (chessBoard[r - j] [c - k * 2 {
oldPiece = chessBoard[r - j][c - k * 2;
if ((Character · isUpperCase (chessBoard[r - j * 2]
· charAt (0) |||"" · equals (chessBoard[r - j * 2]
&&"K" · equals (chessBoard[r - j * 2] [c - k {
oldPiece = chessBoard[r - j * 2][c - k;
chessBoard[r[c = "";
chessBoard[r - j * 2][c - k = "n";
if (King · kingSafeB ()) {
list = list + r + c + (r - j * 2) + (c - k) + oldPiece;

```

```

}
chessBoard[r][c = "n";
chessBoard[r - j * 2][c - k = oldPiece;
}
} catch (Exception e) {
}
}
}
return list;
}
}

```

Bishop.java class

```

package ChessGame;
import static ChessGame.MainClass.chessBoard;
*
* @author a b h a r
*
public class Bishop {
public static String posibleBW(int i) {
String list = "", oldPiece;
int r = i
int temp = 1;
for (int j = -1; j <= 1; j += 2) {
for (int k = -1; k <= 1; k += 2) {
try {
while ("".equals(chessBoard[r + temp * j][c + temp * k]) {
oldPiece = chessBoard[r + temp * j][c + temp * k];
chessBoard[r][c = "";
chessBoard[r + temp * j][c + temp * k = "B";
if (King.kingSafeW()) {
list = list + r + c + (r + temp * j) + (c + temp * k) + oldPiece;
}
chessBoard[r][c = "B";
chessBoard[r + temp * j][c + temp * k = oldPiece;
temp++;
}

```

```

}
if (Character · isLowerCase (chessBoard[r + temp * j]
·charAt (0)
&&"k" · equals (chessBoard[r + temp * j] [c + temp * k {
oldPiece = chessBoard[r + temp * j][c + temp * k;
chessBoard[r][c = "";
chessBoard[r + temp * j][c + temp * k = "B";
if (King · kingSafeW ()) {
list = list + r + c + (r + temp * j) + (c + temp * k) + oldPiece;
}
chessBoard[r][c = "B";
chessBoard[r + temp * j][c + temp * k = oldPiece;
}
} catch (Exception) {
}
temp = 1;
}
}
return list;
}
}
chessBoard[r][c = "b";
chessBoard[r - temp * j][c - temp * k = oldPiece;
temp++;
}
if ((Character · isUpperCase (chessBoard[r - temp * j]
·charAt (0) &&"K" · equals
(chessBoard[r - temp * j] [c - temp * k {
oldPiece = chessBoard[r - temp * j][c - temp * k;
chessBoard[r][c = "";
chessBoard[r - temp * j][c - temp * k = "b";
if (King · kingSafeB ()) {
list = list + r + c + (r - temp * j) + (c - temp * k) + oldPiece;
}
chessBoard[r][c = "b";

```

```

chessBoard[r - temp * j][c - temp * k] = oldPiece;
}
} catch (Exceptione) {
}
temp = 1;
}
public class Rook {
public static String posibleRDW(int i) {
String r = "D";
String list = rookW(r, i);
return list;
}
public static String posibleREW(int i) {
String r = "E";
String list = rookW(r, i);
return list;
}
public static String rookW(String rook, int i) {
String R = rook;
String list = "", oldPiece;
int r = i
int temp = 1;
for (int j = -1; j <= 1; j += 2) {
try {
while ("".equals(chessBoard[r][c + temp * j]) {
oldPiece = chessBoard[r][c + temp * j];
chessBoard[r][c] = "";
chessBoard[r][c + temp * j] = R;
if (King.kingSafeW()) {
list = list + r + c + r + (c + temp * j) + oldPiece;
}
chessBoard[r][c] = R;
chessBoard[r][c + temp * j] = oldPiece; temp++;
}
if (Character.isLowerCase(chessBoard[r][c + temp * j]) .

```

```

charAt(0) &&"k" . equals
(chessBoard[r][c + temp * j {
oldPiece = chessBoard[r][c + temp * j;
chessBoard[r][c = "";
chessBoard[r][c + temp * j = R;
if (King . kingSafeW ()) {
list = list + r + c + r + (c + temp * j) + oldPiece;
}
chessBoard[r][c = R;
chessBoard[r + temp * j][c = oldPiece;
temp++;
}
if (Character . isLowerCase (chessBoard[r + temp * j][c] .
charAt(0) &&"k" . equals
(chessBoard[r + temp * j][c {
oldPiece = chessBoard[r + temp * j][c;
chessBoard[r][c = "";
chessBoard[r + temp * j][c = R;
if (King . kingSafeW ()) {
list = list + r + c + (r + temp * j) + c + oldPiece;
}
chessBoard[r][c = R;
chessBoard[r + temp * j][c = oldPiece;
}
} catch (Exception) {
}
temp = 1;
}
return list;
}
public static String posibleREB(inti) {
String r = "e";
String list = rookB(r, i);
return list;
}

```

```

public static String posibleRDB(inti) {
String r = "d";
String list = rookB(r, i);
return list;
}
public static String rookB(Stringrook, inti) {
String R = rook;
String list = "", oldPiece;
while ("".equals(chessBoard[r + temp * j][c]) {
oldPiece = chessBoard[r + temp * j][c];
chessBoard[r][c] = "";
chessBoard[r + temp * j][c] = R;
if (King.kingSafeB()) {
list = list + r + c + (r + temp * j) + c + oldPiece;
}
chessBoard[r][c] = R;
chessBoard[r + temp * j][c] = oldPiece;
temp++;
}
if (Character.isUpperCase(chessBoard[r + temp * j]
.charAt(0) && "K".equals(chessBoard[r + temp * j]
{
}
} catch (Exception e) {
}
temp = 1;
}
return list;
}
}

```

pawn.java class

```

package ChessGame;
import static ChessGame.MainClass.chessBoard;
*
* @author a b h a r

```

```

*
public class Pawn {
static boolean statePT = true, statePS = true, statePU = true,
statePV = true,
statePW = true, statePX = true, statePY = true, statePZ = true;
static boolean statepT = true, statepS = true, statepU = true,
statepV = true,
statepW = true, statepX = true, statepY = true, statepZ = true;
static boolean qu = true;
static int jState;
public static String posiblePSW(inti) {
String p = "S";
String list = pawnW(p, i) ;
return list;
}
public static String posiblePTW(inti) {
String p = "T";
String list = pawnW(p, i) ;
return list;
}
public static String posiblePUW(inti) {
String p = "U";
String list = pawnW(p, i) ;
return list;
}
}
public static String pawnW(String pawn, inti) {
String P = pawn;
String list = "", oldPiece;
int r = i
for (int j = -1; j <= 1; j+ = 2) {
try {
if (Character.isLowerCase (chessBoard[r - 1] [c + j] ·
charAt(0) && i >= 16) {
oldPiece = chessBoard[r - 1][c + j;

```

```

chessBoard[r][c] = "";
chessBoard[r - 1][c + j] = P;
if (King chessBoard[r - 1][c + j] == "");
}
if ("".equals(chessBoard[r - 1][c + j] && "s").
equals(chessBoard[r][c + j] && statepS == false {
oldPiece = chessBoard[r - 1][c + j];
chessBoard[r][c] = "";
chessBoard[r][c + j] = "";
chessBoard[r - 1][c + j] = P;
jState = j;
if (King.kingSafeW()) {
list = list + r + c + (r - 1) + (c + j) + oldPiece;
}
chessBoard[r][c] = P;
chessBoard[r][c + j] = "s";
chessBoard[r - 1][c + j] = "";
}
if ("".equals(chessBoard[r - 1][c + j] && "u").
equals(chessBoard[r][c + j] && statepU == false {
oldPiece = chessBoard[r - 1][c + j];
chessBoard[r][c] = "";
chessBoard[r][c + j] = "";
chessBoard[r - 1][c + j] = P;
jState = j;
if (King.kingSafeW()) {
}
if ("".equals(chessBoard[r - 1][c + j] && "x").
equals(chessBoard[r][c + j] && statepX == false {
oldPiece = chessBoard[r - 1][c + j];
chessBoard[r][c] = "";
chessBoard[r][c + j] = "";
chessBoard[r - 1][c + j] = P;
jState = j;
if (King.kingSafeW()) {

```



```

list = list + r + c + (r - 1) + (c + j) + oldPiece;
}
chessBoard[r][c] = P;
chessBoard[r][c + j] = "x";
chessBoard[r - 1][c + j] = "";
} if ("" · equals(chessBoard[r - 1][c + j] && "y" ·
equals(chessBoard[r][c + j] && statepY == false {
oldPiece = chessBoard[r - 1][c + j];
chessBoard[r][c] = "";
chessBoard[r][c + j] = "";
chessBoard[r - 1][c + j] = P;
jState = j;
if (King · kingSafeW ()) {
list = list + r + c + (r - 1) + (c + j) + oldPiece;
}
chessBoard[r][c] = P;
chessBoard[r][c + j] = "y";
chessBoard[r - 1][c + j] = "";
chessBoard[r - 1][c + j] = "";
}
} catch (Exception) {
}
try {
if (Character · isLowerCase(chessBoard[r - 1][c + j] ·
charAt(0) && i < 16 {
String[temp] = {"Q", "D", "B", "N", "S"};
for (int k = 0; k < 5; k++) {
oldPiece = chessBoard[r - 1][c + j];
chessBoard[r][c] = "";
chessBoard[r - 1][c + j] = temp[k];
if (King · kingSafeW ()) {
captured-piece,new-piece,P
list = list + c + (c + j) + oldPiece + temp[k] + P;
}
chessBoard[r][c] = P;

```

```

chessBoard[r - 1][c + j] = oldPiece;
}
}
} catch (Exceptione) {
}
}
try {
if ("" · equals(chessBoard[r - 1][c] && i >= 16 {
oldPiece = chessBoard[r - 1][c];
chessBoard[r][c] = "";
chessBoard[r - 1][c] = P;
if (King · kingSafeW()) {
list = list + r + c + (r - 1) + c + oldPiece;
}
chessBoard[r][c] = P;
chessBoard[r - 1][c] = oldPiece;
}
} catch (Exceptione) {
}
try {
if ("" · equals(chessBoard[r - 1][c] && i < 16 {
chessBoard[r][c] = P;
chessBoard[r - 2][c] = oldPiece;
}
} catch (Exceptione) {
}
return list;
}
public static String posiblePSB(inti) {
String p = "s";
String list = pawnB(p, i);
return list;
}
public static String posiblePTB(inti) {
String p = "t";

```

```

String list = pawnB(p, i);
return list;
}
public static String posiblePUB(inti) {
String p = "u";
String list = pawnB(p, i);
return list;
}
public static String posiblePVB(inti) {
String p = "v";
String list = pawnB(p, i);
return list;
}
public static String posiblePWB(inti) {
String p = "w";
String list = pawnB(p, i);
return list;
}
public static String posiblePXB(inti) {
oldPiece = chessBoard[r + 1][c - j]; chessBoard[r][c] = "";
chessBoard[r + 1][c - j] = p;
if (King · kingSafeB ()) {
list = list + r + c + (r + 1) + (
chessBoard[r][c] = p;
chessBoard[r][c - j] = "T";
chessBoard[r + 1][c - j] = ""; }
if (" " · equals (chessBoard[r + 1][c - j] && "S" ·
equals(chessBoard[r][c - j] && statePS == false {
oldPiece = chessBoard[r + 1][c - j];
chessBoard[r][c] = " ";
chessBoard[r][c - j] = " ";
chessBoard[r + 1][c - j] = p;
jState = j;
if (King · kingSafeB ()) {
list = list + r + c + (r + 1) + (c - j) + oldPiece;

```

```

}
chessBoard[r][c = p;
chessBoard[r][c - j = "S";
chessBoard[r + 1][c - j = ""];
}
if ("".equals(chessBoard[r + 1][c - j] && "U").
equals(chessBoard[r][c - j] && statePU == false {
oldPiece = chessBoard[r + 1][c - j;
chessBoard[r][c - j = ""];
chessBoard[r + 1][c - j = p;
jState = j;
if (King.kingSafeB()) {
list = list + r + c + (r + 1) + (c - j) + oldPiece;
}
chessBoard[r][c = p;
chessBoard[r][c - j = "X";
chessBoard[r + 1][c - j = ""];
}
if ("".equals(chessBoard[r + 1][c - j] && "Y").
equals(chessBoard[r][c - j] && statePY == false {
oldPiece = chessBoard[r + 1][c - j;
chessBoard[r][c = ""];
chessBoard[r][c - j = ""];
chessBoard[r + 1][c - j = p;
jState = j;
if (King.kingSafeB()) {
list = list + r + c + (r + 1) + (c - j) + oldPiece;
}
chessBoard[r][c = p;
chessBoard[r][c - j = "Y";
chessBoard[r + 1][c - j = ""];
}
if ("".equals(chessBoard[r + 1][c - j] && "Z").
equals(chessBoard[r][c - j] && statePZ == false {
oldPiece = chessBoard[r + 1][c - j; chessBoard[r][c = ""];

```

```

for (int k = 0; k < 4; k++) {
oldPiece = chessBoard[r + 1][c + j];
chessBoard[r][c] = "";
chessBoard[r + 1][c + j] = temp[k];
if (King.kingSafeB()) {
captured-piece,new-piece,P
list = list + c + (c + j) + oldPiece + temp[k] + p;
}
chessBoard[r][c] = p;
chessBoard[r + 1][c + j] = oldPiece;
}
}
} catch (Exception e) {
}
}
try {
if ("" . equals (chessBoard[r + 1][c] && i >= 8 && i < 48 {
oldPiece = chessBoard[r + 1][c];
chessBoard[r][c] = "";
chessBoard[r + 1][c] = p;
if (King.kingSafeB()) {
list = list + r + c + (r + 1) + c + oldPiece;
} chessBoard[r][c] = p;
chessBoard[r + 1][c] = oldPiece;
}
} catch (Exception e) {
}
try {
if ("" . equals (chessBoard[r + 1][c] && i > 47 {
String[temp] = {"q","d","b","n"};
for (int k = 0; k < 4; k++) {
oldPiece = chessBoard[r + 1][c];
chessBoard[r][c] = "";
chessBoard[r + 1][c] = temp[k];
ht[c];

```

```

chessBoard[r][c = ""];
chessBoard[r + 2][c = p;
if (King · kingSafeB ()) {
list = list + r + c + (r + 2) + c + oldPiece;
}
chessBoard[r][c = p;
chessBoard[r + 2][c = oldPiece;
}
} catch (Exceptione) {
}
return list;
}
}

```

Interface.java class

```

package ChessGame;
import java·awt · *;
import java·awt · event · *;
import javax·swing · *;
*
* @author a b h a r
*
public class InterFace extends JPanel implements ActionListener {
static boolean info = false;
static String infoAdd;
InterFace thisFrame = this;
static boolean state = true;
Timer tm = new Timer(10, this);
int aniV1 = -630, aniV for (inti = 0; i < 64; i + = 2) {
g·setColor (newColor (red, green, blue));
g·fillRect ((i%8 + (i
squareSize + sizeY, squareSize, squareSize;
g·setColor (newColor (255, 255, 255));
g·drawRect ((i%8 + (i
squareSize + sizeY, squareSize, squareSize;
Image im1 = new ImageIcon(getClass () · getClassLoader () ·
```

```

getResource("image
g.drawImage(im1, ((i + 1) % 8 - ((i + 1)
squareSize + sizeX, ((i + 1)
g.setColor(newColor(255, 255, 255));
g.drawRect(((i + 1) % 8 - ((i + 1)
squareSize + sizeX, ((i + 1)
g.setColor(newColor(255, 255, 255));
g.drawRect((i % 8 + (i
(i}
} } else if (King.kingSafeB()) {
int i = MainClass.kingPositionB;
for (int q = 0; q < 64; q++ = 2) {
if (i == q) {
g.drawImage(im1, (i % 8 + (i
(i
g.setColor(newColor(255, 255, 255));
g.drawRect((i % 8 + (i
(i}
}
for (int l = 1; l < 64; l++ = 2) {
if (i == l) {
g.drawImage(im1, (i % 8 + (i
(ig.setColor(newColor(255, 255, 255));
g.drawRect((i % 8 + (i
(i}
}
}
Image chessPieceImage;
chessPieceImage = new ImageIcon(getClass().getClassLoader().
getResource("image
for (int i = 0; i < 64; i++) {
case "v":
case "w":
case "x":
case "y":

```

```

case "z":
j = 5;
k = Algorithm·humanAsWhite;
k = Algorithm·humanAsWhite;
break;
case "K":
j = 0;
k = 1 - Algorithm·humanAsWhite;
break;
case "k":
j = 0;
k = Algorithm·humanAsWhite;
break;
}
if (j = -1&& k = -1) {
g·drawImage(chessPieceImage, (i%8) * squareSize + sizeX +
mouseDrag[i
sizeY + mouseDrag[i
mouseDrag[i
sizeY + mouseDrag[i
j = 4;
k = 1;
break;
case "b":
j = 3;
k = 1;
break;
break;
case "B":
j = 3;
k = 0;
break;
case "Q":
j = 1;
k = 0;

```



```

break;
case "K":
j = 0;
k = 0;
break;
}
if (j = -1 && k = -1) {
g.drawImage(chessPieceImage, (i%2) * squareSize + sizeX - 2
squareSize - 10, (i
, (i%2 + 1) * squareSize + sizeX - 2 * squareSize - 10, (i
squareSize + sizeY, j * 64, k * 64, (j + 1) * 64, (k + 1) * 64, this;
}
}
if (info == true) {
Image image1 = new ImageIcon(getClass().getClassLoader().
getResource("image
int heightSize = image1.getHeight(this) + 5;
g.drawImage(image1, 5, getHeight() - heightSize, this);
}
}
package ChessGame;
import static ChessGame.MainClass.chessBoard;
*
* @author a b h a r
*
public class Queen {
public static String posibleQW(int i) {
String list = "", oldPiece;
int r = i
int temp = 1;
for (int j = -1; j <= 1; j++) {
for (int k = -1; k <= 1; k++) {
if (j = 0 || k = 0) {
try {
while ("".equals(chessBoard[r + temp * j][c + temp * k])) {

```

```

oldPiece = chessBoard[r + temp * j][c + temp * k];
chessBoard[r][c] = "";
chessBoard[r + temp * j][c + temp * k] = "Q";
if (King.kingSafeW()) {
list = list + r + c + (r + temp * j) + (c + temp * k) + oldPiece;
}
chessBoard[r][c] = "Q";
chessBoard[r + temp * j][c + temp * k] = oldPiece;
temp++;
}
if (Character.isLowerCase(chessBoard[r + temp * j]
[c + temp * k].charAt(0)) {
oldPiece = chessBoard[r + temp * j][c + temp * k];
chessBoard[r][c] = "";
chessBoard[r + temp * j][c + temp * k] = "Q";
if (King.kingSafeW()) {
list = list + r + c + (r + temp * j) + (c + temp * k) + oldPiece;
}
chessBoard[r][c] = "Q";
chessBoard[r + temp * j][c + temp * k] = oldPiece;
}
} catch (Exceptione) {
}
temp = 1;
}
}
}
return list;
}
public static String posibleQB(int i) {
String list = "", oldPiece;
int r = i
int temp = 1;
for (int j = -1; j <= 1; j++) {
for (int k = -1; k <= 1; k++) {

```

```

if (j = 0 || k = 0) {
try {
while (" " . equals (chessBoard[r - temp * j][c - temp * k])) {
oldPiece = chessBoard[r - temp * j][c - temp * k];
chessBoard[r][c] = " ";
chessBoard[r - temp * j][c - temp * k] = "q";
if (King . kingSafeB()) {
list = list + r + c + (r - temp * j) + (c - temp * k) + oldPiece;
}
chessBoard[r][c] = "q";
}
return list;
}
}

```

Interface

```

package ChessGame;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
*
* @author a b h a r
*
public class InterFace extends JPanel implements ActionListener {
static boolean info = false;
static String infoAdd;
InterFace thisFrame = this;
static boolean state = true;
Timer tm = new Timer(10, this);
int aniV1 = -630, aniV2 = 0, aniV3 = 4, aniV5 = -10, aniV6 = 589;
static int mouseX = 50, mouseY = 50, newMouseX, newMouseY;
static int squareSize = 70;
static int mouseDrag[][][] = newint[8][8][2];
static int border = 20;
}
};

```

```

Image im11 = new ImageIcon(getClass().getClassLoader().
getResource("image
g.drawImage(im11,0,0,getWidth(),getHeight(),this);
Image im12 = new ImageIcon(getClass().getClassLoader().
getResource("image
;
g.drawImage(im12,(0%8+(0
squareSize+sizeY-18,8* squareSize+33,8* squareSize+47,this;
g.setColor(newColor(250,245,162));
g.fillRect((0%8+(0
squareSize+sizeY-3,8* squareSize+7,8* squareSize+7;
for (inti=0;i<64;i+=2){
g.setColor(newColor(red,green,blue));
g.fillRect((i%8+(i
squareSize+sizeY,squareSize,squareSize;
g.setColor(newColor(255,255,255));
g.drawRect((i%8+(i
squareSize+sizeY,squareSize,squareSize;
Image im1 = new ImageIcon(getClass().getClassLoader().
getResource("image
g.drawImage(im1,((i+1)%8-((i+1)
for (intl=1;l<64;l+=2){
if (i==l){
g.drawImage(im1,(i%8+(i
(i
g.setColor(newColor(255,255,255));
g.drawRect((i%8+(i
(i}
} } else if (King.kingSafeB()){
int i = MainClass.kingPositionB;
for (intq=0;q<64;q+=2){
if (i==q){
g.drawImage(im1,(i%8+(i
(i
g.setColor(newColor(255,255,255));

```

```

g.drawRect ((i%8 + (i
(i}
}
for (intl = 1; l < 64; l+ = 2) {
if (i == l) {
g.drawImage (im1, (i%8 + (i
(ig · setColor (newColor (255, 255, 255));
g.drawRect ((i%8 + (i
(i}
}
}
Image chessPieceImage;
chessPieceImage = new ImageIcon(getClass () · getClassLoader () ·
getResource("image
for (inti = 0; i < 64; i + +) {
case "u":
case "v":
case "w":
case "x":
case "y":
case "z":
j = 5;
k = Algorithm.humanAsWhite;
break;
case "D":
case "E":
j = 2;
k = 1 - Algorithm.humanAsWhite;
break;
case "d":
case "e":
j = 2;
k = Algorithm.humanAsWhite;
break;
case "N":

```

```

j = 4;
k = 1 - Algorithm·humanAsWhite;
break;
case "n":
j = 4;
k = Algorithm·humanAsWhite;
break;
case "B":
j = 3;
k = 1 - Algorithm·humanAsWhite;
break;
case "b":
j = 3;
k = Algorithm·humanAsWhite;
break;
case "Q":
j = 1;
k = 1 - Algorithm·humanAsWhite;
break;
case "q":
j = 1;
k = Algorithm·humanAsWhite;
break;
case "K":
j = 0;
k = 1 - Algorithm·humanAsWhite;
break;
case "k":
j = 0;
k = Algorithm·humanAsWhite;
break;
}
if (j = -1 && k = -1) {
g·drawImage(chessPieceImage, (i%8) * squareSize + sizeX +
mouseDrag[i

```

```

sizeY + mouseDrag[i
mouseDrag[i
sizeY + mouseDrag[i
}
Image image = new ImageIcon(getClass () . getClassLoader () .
getResource("image
Image image1 = new ImageIcon(getClass () . getClassLoader () .
getResource("image
g.drawImage (image, aniV2, 0, getWidth () , getHeight () , this) ;
j = 5;
k = 1;
break;
case "d":
case "e":
j = 2;
k = 1;
break;
case "n":
j = 4;
k = 1;
break;
case "b":
j = 3;
k = 1;
break;
case "q":
j = 1;
k = 1;
break;
case "k":
j = 0;
k = 1;
break;
}
if (j = -1&& k = -1) {

```

```

g.drawImage(chessPieceImage, (i%2) * squareSize + sizeX + 8*
squareSize + 10, (i
squareSize + +sizeX + 8 * squareSize + 10, (i
squareSize + sizeY, j * 64, k * 64, (j + 1) * 64, (k + 1) * 64, this;
}
}
for (inti = 0;
i < 16; i++){
int j = -1, k = -1;
switch (MainClass · caputerW[i
case "S":
case "T":
case "U":
case "V":
case "W":
case "X":
case "Y":
case "Z":
j = 5;
k = 0;
break;
case "D":
case "E":
j = 2;
k = 0;
break;
case "N":
j = 4;
k = 0;
break;
case "B":
j = 3;
k = 0;
import static ChessGame·MainClass · makeMoveB;
import static ChessGame·MainClass · undoMoveB;

```



```

import static ChessGame.King.kingSafeB;
import static ChessGame.King.possibleKB;
*
* @author a b h a r
*
public class Algorithm {
static int humanAsWhite = 1;
as white , 0= human as black
static int globalDepth = 1;
static int pawnBoard[] [] = {
{0,0,0,0,0,0,0,0},
{50,50,50,50,50,50,50,50},
{10,10,20,30,30,20,10,10},
{5,5,10,25,25,10,5,5},
{0,0,0,20,20,0,0,0},
{5,-5,-10,0,0,-10,-5,5},
{5,10,10,-20,-20,10,10,5},
{0,0,0,0,0,0,0,0}};
static int rookBoard[] [] = {
{0,0,0,0,0,0,0,0},
{5,10,10,10,10,10,10,5},
{-5,0,0,0,0,0,0,-5},
{-5,0,0,0,0,0,0,-5},
{-5,0,0,0,0,0,0,-5},
{-5,0,0,0,0,0,0,-5},
{-5,0,0,0,0,0,0,-5},
{0,0,0,5,5,0,0,0}};
static int knightBoard[] [] = {
{-50,-40,-30,-30,-30,-30,-40,-50},
{-40,-20,0,0,0,0,-20,-40},
{-30,0,10,15,15,10,0,-30},
{-30,5,15,20,20,15,5,-30},
{-30,0,15,20,20,15,0,-30},
{-30,5,10,15,15,10,5,-30},
{-40,-20,0,5,5,0,-20,-40},

```

```

{-50, -40, -30, -30, -30, -30, -40, -50}};
static int bishopBoard[] [] = {
{-20, -10, -10, -10, -10, -10, -10, -20},
{-10, 0, 0, 0, 0, 0, 0, -10},
{-10, 0, 5, 10, 10, 5, 0, -10},
{-10, 5, 5, 10, 10, 5, 5, -10},
{-10, 0, 10, 10, 10, 10, 0, -10},
{-10, 10, 10, 10, 10, 10, 10, -10},
{-10, 5, 0, 0, 0, 0, 5, -10},
{-20, -10, -10, -10, -10, -10, -10, -20}};
static int queenBoard[] [] = {
{-20, -10, -10, -5, -5, -10, -10, -20},
{-10, 0, 0, 0, 0, 0, 0, -10},
{-10, 0, 5, 5, 5, 5, 0, -10},
{-5, 0, 5, 5, 5, 5, 0, -5},
{0, 0, 5, 5, 5, 5, 0, -5},
{-10, 5, 5, 5, 5, 5, 0, -10},
{-10, 0, 5, 0, 0, 0, 0, -10},
{-20, -10, -10, -5, -5, -10, -10, -20}};
static int kingMidBoard[] [] = {
String returnString = alphaBeta(depth - 1, beta, alpha,
list.substring(i, i + 5), player; intValue = Integer.valueOf(returnString.substring
(5, returnString.length()));
undoMoveB(list.substring(i, i + 5));
if (player == 0) {
if (value <= beta) {
beta = value;
if (depth == globalDepth) {
move = returnString.substring(0, 5);
}
}
} else {
if (value > alpha) {
alpha = value;
if (depth == globalDepth) {

```

```

move = returnString.substring(0, 5);
}
}
}
if (alpha >= beta) {
if (player == 0) {
return move + beta;
} else {
return move + alpha;
}
}
}
if (player == 0) {
return move + beta;
} else {
return move + alpha;
}
}
public static String sortMoves(Stringlist) {
int[] score = newint[list.length()]
for (inti = 0; i < list.length(); i += 5) {
makeMoveB(list.substring(i, i + 5));
score[i]
undoMoveB(list.substring(i, i + 5));
}
String newListA = "", newListB = list;
for (inti = 0; i < Math.min(6, list.length()); i++) {
int max = -1000000, maxLocation = 0;
for (intj = 0; j < list.length())
if (score[j] > max) {
max = score[j];
maxLocation = j;
}
}
score[maxLocation] = -1000000;

```

```

newListA += list.substring(maxLocation * 5, maxLocation * 5 + 5);
newListB += newListB.replace(list.substring
(maxLocation * 5, maxLocation * 5 + 5), "");}
return newListA + newListB;
}
public static int rating(intlist, intdepth) {
int counter = 0, material = rateMaterial();
kingPositionB = i;
if (!kingSafeB()) {
counter -= 100;
}
break;
case "e":
case "d":
kingPositionB = i;
if (!kingSafeB()) {
counter -= 500;
}
break;
case "n":
kingPositionB = i;
if (!kingSafeB()) {
counter -= 300;
}
break;
case "b":
kingPositionB = i;
if (!kingSafeB()) {
counter -= 300;
}
break;
case "q":
kingPositionB = i;
if (!kingSafeB()) {
counter -= 900;
}

```

```

}
break;
}
}
kingPositionB = tempPositionL;
if (!kingSafeB()) {
counter -= 400;
}
return counter
}
public static int rateMaterial() {
int counter = 0, bishopCounter = 0;
for (inti = 0; i < 64; i++) {
switch (chessBoard[i
bishopCounter += 1;
break;
case "q":
counter += 900;
break; }
if (bishopCounter >= 2) {
counter += 300 * bishopCounter;
} else {
if (bishopCounter == 1) {
}
}
return counter;
}
}

```