

DAYANANDA SAGAR UNIVERSITY

School of Engineering, Kudlu Gate

Bangalore-560068



CERTIFICATE

This is to certify that Mr./Ms. KUSHAL N bearing USN ENG17CS0115 has satisfactorily completed his/her Mini Project as prescribed by the University for the 4TH semester B.Tech. programme in Computer Science & Engineering during the year 2018-19 at the School of Engineering, Dayananda Sagar University., Bangalore.

Date: _____

Signature of the faculty in-charge

Max Marks	Marks Obtained

Signature of Chairman
Department of Computer Science & Engineering

DECLARATION

We hereby declare that the work presented in this mini project entitled-
“ Chess using Minimax AI algorithm ”, has been carried out by us and it
has not been submitted for the award of any degree, diploma or the mini
project of any other college or university.

KUSHAL N (ENG17CS0115)

ACKNOWLEDGEMENT

The satisfaction that accompanies the successful completion of task would be incomplete without the mention of the people who made it possible and whose constant guidance and encouragement crown all the efforts with success.

We are especially thankful to our **Chairman ,Dr. M K Banga**,for providing necessary departmental facilities, moral support and encouragement.

We are very much thankful to **Prof.Kavyashree** ma'am, for providing help and suggestions in completion of this mini project successfully.

We have received a great deal of guidance and co-operation from our friends and we wish to thank all that have directly or indirectly helped us in the successful completion of this project work.

KUSHAL N (ENG17CS0115)

TABLE OF CONTENTS

<u>Contents</u>	<u>Page no</u>
Introduction	7 - 8
Problem statement	9
Literature review	10 - 11
Hardware and Software requirements	12
Design	
• Pseudo code	13 - 16
• Flowchart	17 - 19
Testing	20 - 21
Results	22 - 26
Conclusion	27
References	28

TABLE OF FIGURES

Figures	Page no
1	10
2	13
3	16
4	17
5	18
6	21
7	22
8	23
9	24
10	25
11	26

ABSTRACT

The Chess game is developed by using Min max Algorithm and Alpha-beta pruning . The approach is more efficient as compared to brute force method of building the chess game . This game is executed on a web browser using JavaScript programming language and JSON .

1. Introduction

The current level of development in computer chess programming is fairly complicated, yet interesting as well. In this project, we were supposed to develop a chess-playing program. The program was supposed to play chess at a good level.

1.1 About the problem

Chess is a two-player strategy board game played on a chessboard, a checkered game board with 64 squares arranged in an 8×8 grid. The game is played by millions of people worldwide. Each player begins with 16 pieces: one king, one queen, two rooks, two knights, two bishops, and eight pawns. Each of the six piece types moves differently, with the most powerful being the queen and the least powerful the pawn. The objective is to checkmate the opponent's king by placing it under an inescapable threat of capture. To this end, a player's pieces are used to attack and capture the opponent's pieces, while supporting each other. In addition to checkmate, the game can be won by voluntary resignation of the opponent, which typically occurs when too much material is lost or checkmate appears inevitable. There are also several ways a game can end in a draw. Chess first appeared in India about the 6th century AD and by the 10th century had spread from Asia to the Middle East and Europe. Since at least the 15th century, chess has been known as the “Royal game” because of its popularity among the nobility. Rules and set design slowly evolved until both reached today's standard in the early 19th century. Once an intellectual diversion favored by the upper classes, chess went through an explosive growth in interest during the 20th century as professional and state-sponsored players competed for an officially recognized world championship title.

1.2 About the ADA technique

Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions, and abandons a candidate ("Backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.

Backtracking can be applied only for problems which admit the concept of a "partial candidate solution" and a relatively quick test of whether it can possibly be completed to a valid solution. It is useless, for example, for locating a given value in an unordered table. When it is applicable, however, backtracking is often much faster than brute force enumeration of all complete candidates, since it can eliminate a large number of candidates with a single test.

Backtracking is an important tool for solving constraint satisfaction problems, such as crosswords, verbal arithmetic, Sudoku, and many other puzzles. It is often the most convenient technique for parsing, for the knapsack problem and other optimization problems.

Minimax is a recursive algorithm which is used to choose an optimal move for a player assuming that the other player is also playing optimally. It is used in games such as tic-tac-toe, go, chess, checkers, and many other two-player games. Such games are called games of perfect information because it is possible to see all the possible moves of a particular game.

There can be two-player games which are not of perfect information such as Scrabble because the opponent's move cannot be predicted.

Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally. It is widely used in two player turn based games such as Tic-Tac-Toe, Backgamon, Mancala, Chess, etc.

In Minimax the two players are called maximizer and minimizer. The maximizer tries to get the highest score possible while the minimizer tries to get the lowest score possible while minimizer tries to do opposite.

Every board state has a value associated with it. In a given state if the maximizer has upper hand then, the score of the board will tend to be some positive value. If the minimizer has the upper hand in that board state then it will tend to be some negative value. The values of the board are calculated by some heuristics which are unique for every type of game.

2. Problem Statement

Generally Chess is being built using Brute – Force algorithm. A **brute-force algorithm** tends to use the simplest possible approach to solving the problem. The major **disadvantage** is that a **brute -force** approach works well only for a small number of nodes.

Disadvantages:

- It is inefficient and hence useless when dealing with homogeneous problems of higher complexity
- Not suitable for solving problems that have an hierarchial structure and involve logical operations.

Proposed Approach:

Alpha–beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. It is an adversarial search algorithm used commonly for machine playing of two-player games (Tic-tac-toe, Chess, Go, etc.). It stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

3. Literature Review

The program is implemented entirely in Java. The following diagram shows the relationships between different modules of the program. Below the diagram is a listing of the modules, as they correspond to Java classes, with short descriptions and links to the source files. The algorithms used are :

1. Min-max Searching
2. Alpha beta pruning

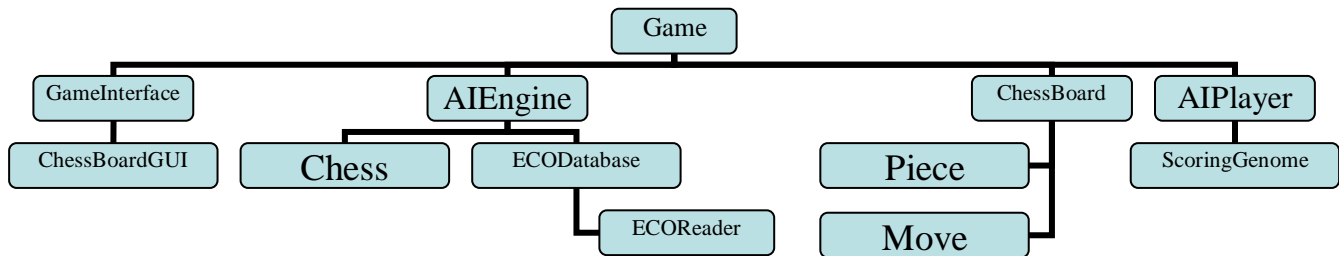


Fig 1: Overall View to show how the Game works

The evaluation function calculates a numeric value for each node of the game tree, we need to apply a certain search strategy to construct an acceptable chess program. At this search step, we are planning to apply minimax search algorithm to choose the best possible next move and alpha-beta pruning technique to increase the performance by decreasing the search space.

The basic idea underlying all two-agent search algorithms is Minimax method. Minimax is a recursive algorithm used for choosing the next move in a game. A tree of legal moves is built and played. Each move is evaluated using an evaluation function. The computer makes the move that maximizes the minimum value of the position resulting from the opponent's possible following moves. The details of our Minimax algorithm will be explained later. The basic idea behind the Alpha-Beta Pruning algorithm is that, once you have a good move, you can quickly eliminate alternatives that lead to disaster. We know that there are many of those quickly disposable moves in chess game. The alpha-beta pruning function is actually an improvement of the Minimax search method. It reduces the number of tree nodes to evaluate by eliminating a move when at least one possibility was proved worse than a previously evaluated one. The details of our alpha-beta pruning algorithm will be explained later.

This project implements a classic version of Chess. The Chess game follows the basic rules of chess, and all the chess pieces only move according to valid moves for that piece. Our implementation of Chess is for One player against Artificial Intelligence as the 2nd player. It is played on an 8x8 checkered board , with a dark square in each player's lower left corner.

Research has been conducted in Chess artificial intelligence and creating a more Realistic and moreover intelligent Chess match. Russell discussed applying an evaluation function to the

leaves of the tree, that judges the value of certain moves from a given position. Another method he mentioned is to cut off the search by setting a limit to its depth. Russell evaluated a particular technique called alpha-beta pruning to remove branches of a tree that will not influence the final decision. Bratko discusses the method of chess players to "chunk" together familiar chess patterns, and using this to reduce the complexity for AI when considering a position. However, this technique is in its early stages, and requires that multiple assumptions and a complicated detection process. Berliner recognized that two similar positions can be very different. However, this kind of research is essentially never complete.

This paper concerns the research and development of an Automated Chess Annotation program - a program which attempts to provide automatic commentary on any chess game it is given. The closest research to chess annotation is that performed by de Groot in 1965, who performed psychological experiments to try to determine how humans thought about concepts in chess, but the interaction between computer chess and human commentary does not appear to have been examined before. Development in computer programs has been dominated by number crunching evaluation over pattern matching, and this has lead development away from human thought patterns and annotation. The program is developed by extending an existing open source chess program called Crafty. Crafty was chosen because it had several attractive features with regards to chess annotation development, especially regarding the free availability of its source code and its strong chess-playing ability. Crafty was, however, found over the course of the project to have several disadvantages with regards to development, in particular concerning the language it is written in and the complexity and interlinking of its code. Chess Annotation is the process of providing commentary on a chess game. Computers play chess by attempting to 'brute-force' examine as many potential moves in as short a time as possible, using a limited pattern matching system and a lot of number crunching to evaluate each move.

Automated annotation must therefore convert the computer's analysis into the concepts a human uses when playing chess. This may take the form of simply putting the computer's values into words (as in "White's last move was very poor"), deriving human-recognizable features from the computer's data (such as *threats*), or even creating new measures to deal with abstract human concepts (such as initiative, psychological advantage, and obviousness). The value of this work is in the insights it provides into the realm of human and machine perception.

4. Software and Hardware Requirements

4.1 Software Requirements:

- **Ubuntu / Windows XP and above**
- **HTML Editors :** Notepad or any similar text Editors
- **Node.js :** Node.js is a JavaScript runtime environment. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient.
- **Sublime Text editor**

4.2 Hardware Requirements:

Laptop or Personal Computer with the given minimum specifications :

- Microsoft Windows 10/8/7/Vista/2003/XP (incl.64-bit)
- 2 GB RAM minimum
- 4 GB RAM recommended
- 1024 x 768 minimum screen resolution
- Intel Pentium IV and above
- Keyboard and Mouse

5. Design

5.1 Pseudo code

There are 5 modules in our program, They are:

- Move generation and Board visualization.
- Position evaluation
- Search Tree using Minimax
- Alpha-Beta pruning

5.1.1 : Move generation and board visualization

We'll use the [chess.js](#) library for move generation, and [chessboard.js](#) for visualizing the board. The move generation library basically implements all the rules of chess. Based on this, we can calculate all legal moves for a given board state. Using these libraries will help us focus only on the most interesting task, creating the algorithm that finds the best move.

Function that just returns a random move from all of the possible moves:

```
var calculateBestMove=function(game)
{
    //generate all the moves for a given position
    var newGameMoves = game.ugly_moves();

    return newGameMoves[Math.floor(Math.random() *
newGameMoves.length)];
}
```

5.1.2 : Position evaluation

Now we should understand which side is stronger in a certain position. The simplest way to achieve this is to count the relative strength of the pieces on the board using the following table.








	10		-10
	30		-30
	30		-30
	50		-50
	90		-90
	900		-900

Fig 2 : Weightage of each chess piece

With the evaluation function, we're able to create an algorithm that chooses the move that gives the highest evaluation

```
var calculateBestMove = function (game)

{

    var newGameMoves = game.ugly_moves();

    var bestMove = null;

    //use any negative large number

    var bestValue = -9999;

    for (var i = 0; i < newGameMoves.length; i++) {

        var newGameMove = newGameMoves[i];

        game.ugly_move(newGameMove);

        //take the negative as AI plays as black

        var boardValue = -evaluateBoard(game.board())

        game.undo();

        if (boardValue > bestValue) {

            bestValue = boardValue;

            bestMove = newGameMove

        }

    }

    return bestMove;

};
```

5.1.3 : Search tree using Minimax

Next we're going to create a search tree from which the algorithm can chose the best move. This is done by using the Minimax algorithm. In this algorithm, the recursive tree of all possible moves is explored to a given depth, and the position is evaluated at the ending "leaves" of the tree. After

that, we return either the smallest or the largest value of the child to the parent node, depending on whether it's a white or black to move. (That is, we try to either minimize or maximize the outcome at each level.)

```
var minimax = function (depth, game, isMaximisingPlayer)
{
    if (depth === 0) {
        return -evaluateBoard(game.board());
    }
    var newGameMoves = game.ugly_moves();
    if (isMaximisingPlayer)
    {
        var bestMove = -9999;
        for (var i = 0; i < newGameMoves.length; i++)
        {
            game.ugly_move(newGameMoves[i]);
            bestMove = Math.max(bestMove, minimax(depth - 1, game, !isMaximisingPlayer));
            game.undo();
        }
        return bestMove;
    }
    else
    {
        var bestMove = 9999;
        for (var i = 0; i < newGameMoves.length; i++)
        {
            game.ugly_move(newGameMoves[i]);
            bestMove = Math.min(bestMove, minimax(depth - 1, game, !isMaximisingPlayer));
            game.undo();
        }
        return bestMove;
    }
};
```

5.1.4 : Alpha-beta pruning

Alpha-beta pruning is an optimization method to the minimax algorithm that allows us to disregard some branches in the search tree. This helps us evaluate the minimax search tree much deeper, while using the same resources.

The alpha-beta pruning is based on the situation where we can stop evaluating a part of the search tree if we find a move that leads to a worse situation than a previously discovered move.

5.1.5 : Improved evaluation function

The initial evaluation function is quite naive as we only count the material that is found on the board. To improve this, we add to the evaluation a factor that takes in account the position of the pieces. For example, a knight on the center of the board is better (because it has more options and is thus more active) than a knight on the edge of the board.

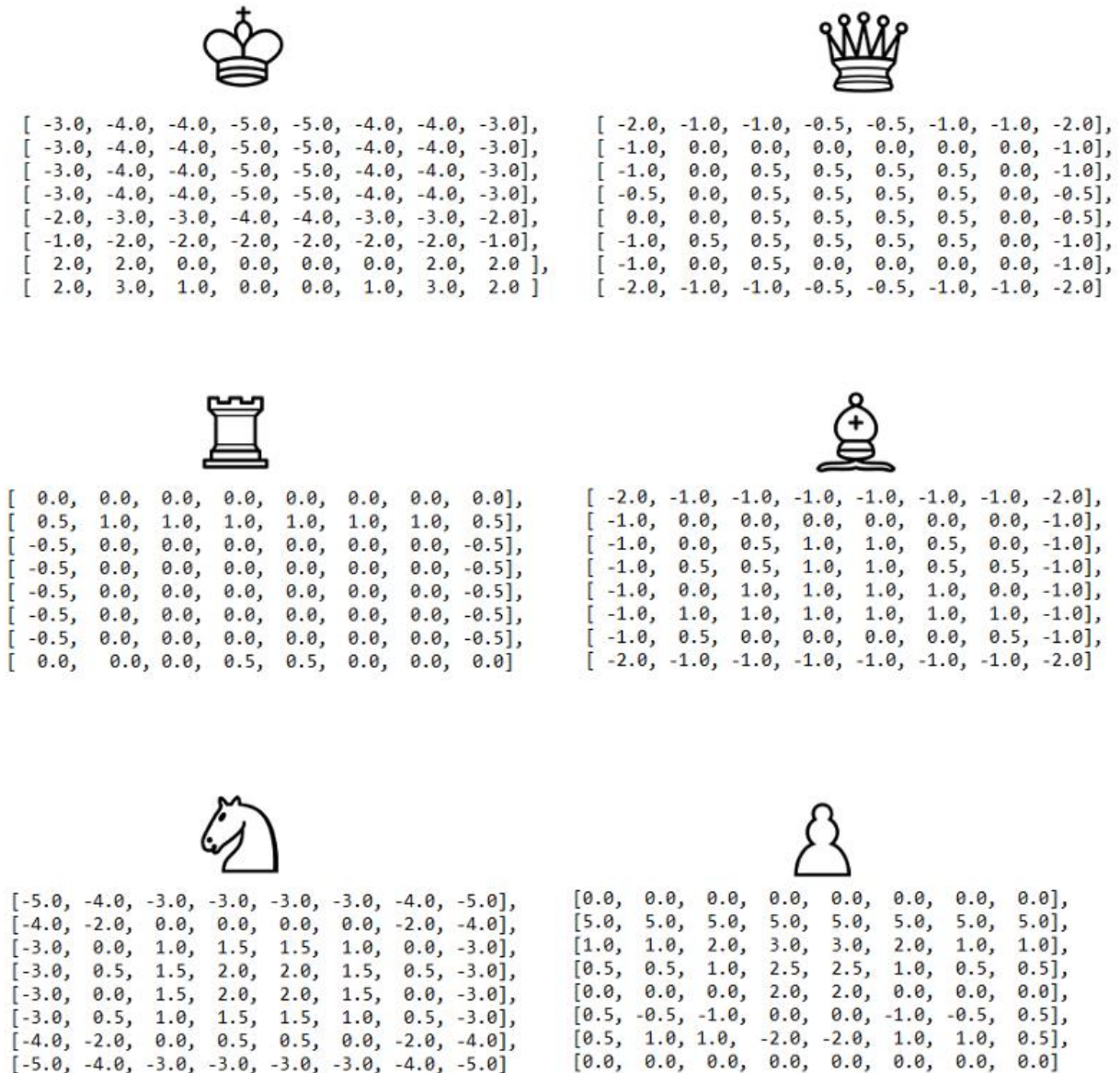


Fig 3 : Move Visualization

5.2 Flow chart

State space tree :

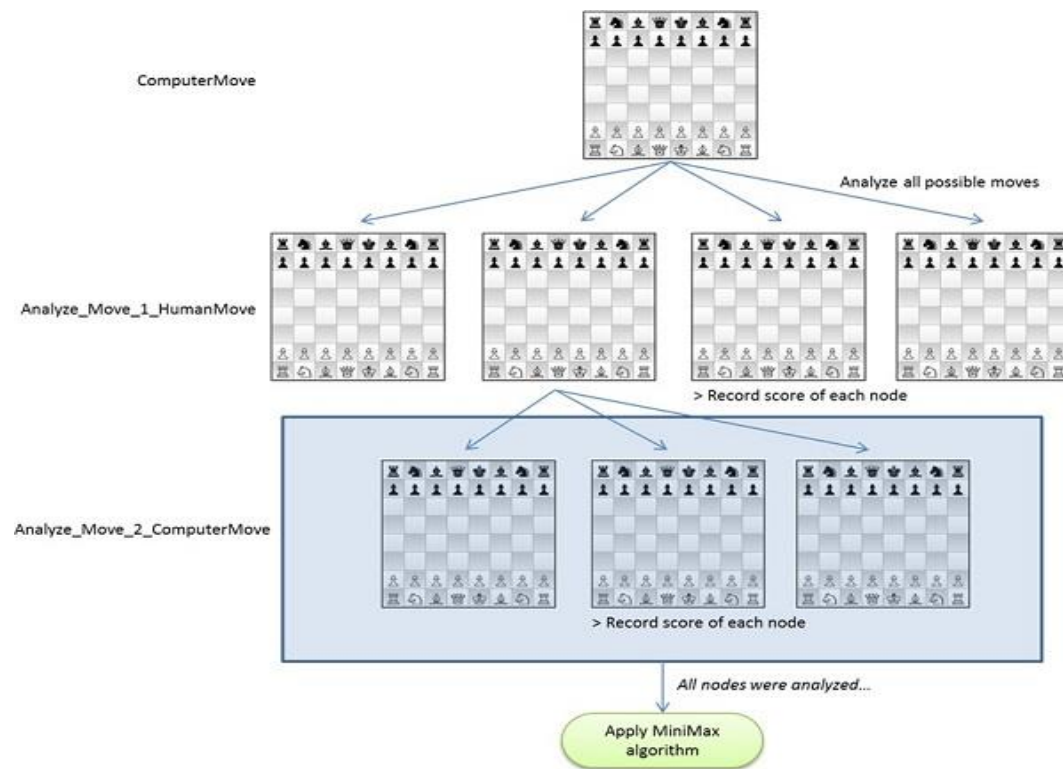


Fig 4 : State Space Tree

Flow chart:

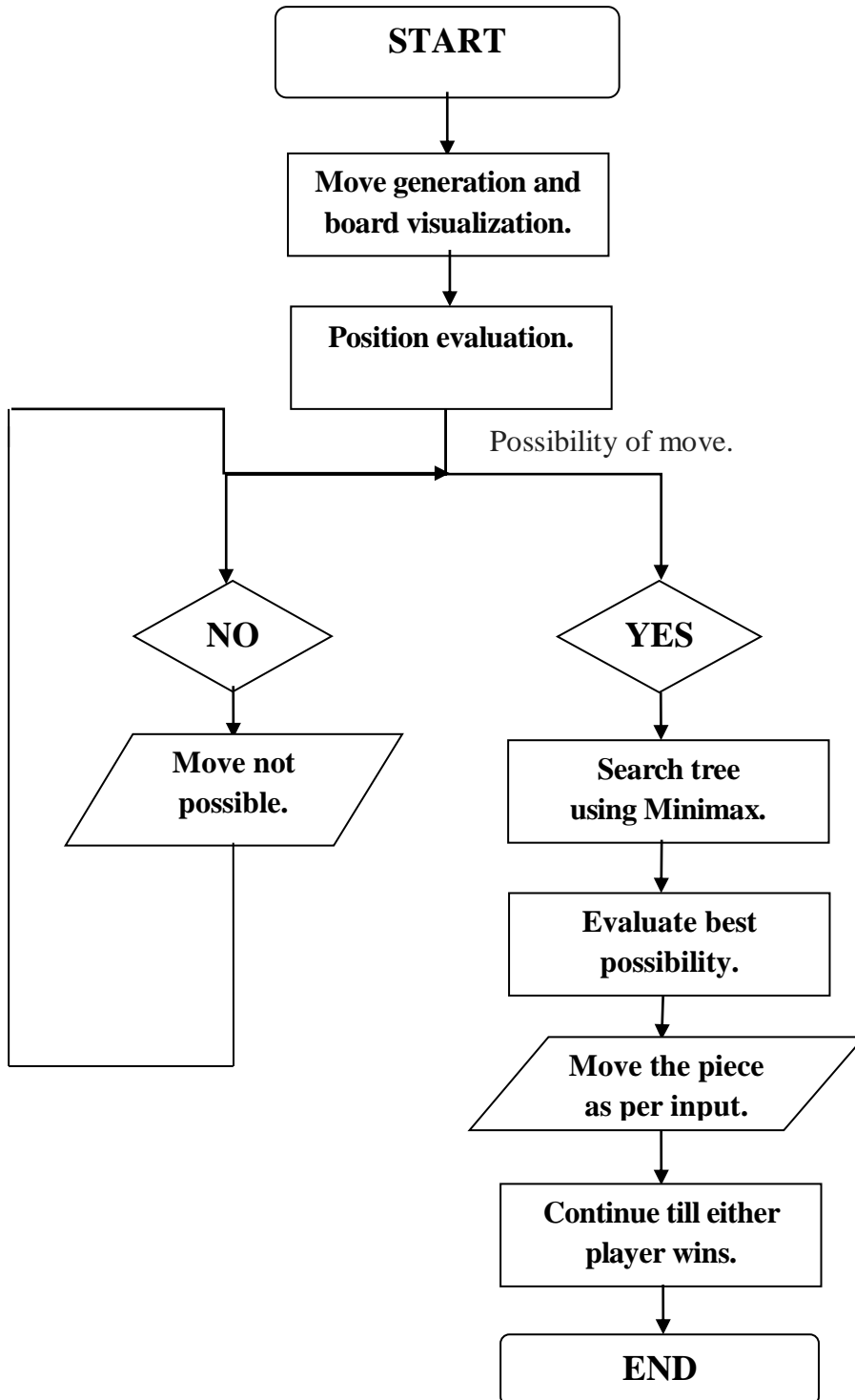


Fig 5 : Schematic flow chart of the chess game

Minimax (sometimes MinMax or MM) is a decision rule used in decision theory, game theory, statistics and philosophy for *minimizing* the possible loss for a worst case (*maximum* loss) scenario. When dealing with gains, it is referred to as "maximin"—to maximize the minimum gain. Originally formulated for two-player zero-sum game theory, covering both the cases where players take alternate moves and those where they make simultaneous moves, it has also been extended to more complex games and to general decision-making in the presence of uncertainty.

6. Testing

Testing software is a critical element of software quality assurance and represents the ultimate review of specification, design and code generation.

Testing Objectives

1. Testing is a process of executing a program with the intent of finding an error.
2. A good test case is one that has a high probability of finding an undiscovered error.

Black box Testing

In black box testing, the system is tested as whole without considering the internal process and their inputs and outputs. Black box tests are used to demonstrate that software functions are operational, that input is properly accepted and output is correctly produced, and that the integrity of external information is maintained.

White box Testing

White box testing is a technique that is done to verify the internal process, their inputs and outputs. Hence the test cases are derived using the control structure of procedural design. The system test reducing the test case design. Each function is executed separately and results are found to be as expected lines. The fields which are compulsory are validated. It is found that all the functions are working satisfactorily according to the specifications.

Sl.No	Input	Expected Output	Actual Output	Success/Fail
1	Opening an index file	The Chess Board should be generated	Chess Board is generated	Success
2	Check whether all the pieces of the both sides are available or not and also check whether they are properly organized or not.	All pieces should be properly placed on the board	All pieces on both the kings and queens side are properly placed in both colors.	Success
3	Check if the moves AI is doing is valid	AI generated output should work clean and properly	AI generated outputs worked properly	Success
4	Check if any Checkmate can be done	Give a checkmate to the opposite player	Checkmate given	Success

Fig 6 : Table containing Inputs and Expected Outputs with success or Failure

7.RESULTS:

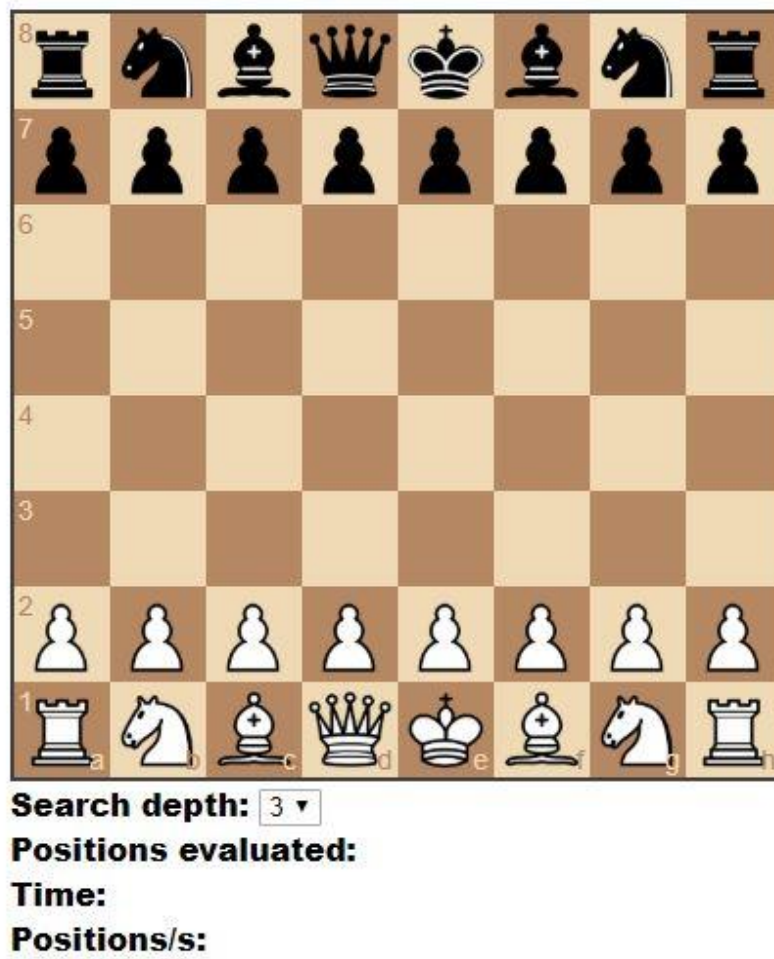


Fig 7 : Initial positions of the icons in the Game.



Search depth: 3 ▼
 Positions evaluated: 9980
 Time: 0.738s
 Positions/s: 13523.035230352303

d4 Nc6
 a3 Nf6
 h3 d5

Fig 8 : Some of the Moves played



Search depth: 3 ▼

Positions evaluated: 18782

Time: 1.919s

Positions/s: 9787.389265242313

Qd5 Ne4

Qb5 Qd6

Qxb7 Rb8

Qxb8+ Nxb8

Kd1 Nxf2+

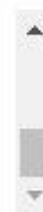


Fig 9 : Intermediate Snapshot of the game



Search depth: 3 ▼

Positions evaluated: 6355

Time: 0.68s

Positions/s: 9345.588235294117

113 03

Nf3 Ne4

b3 f5

h4 Be6

Ng5 Nxg5

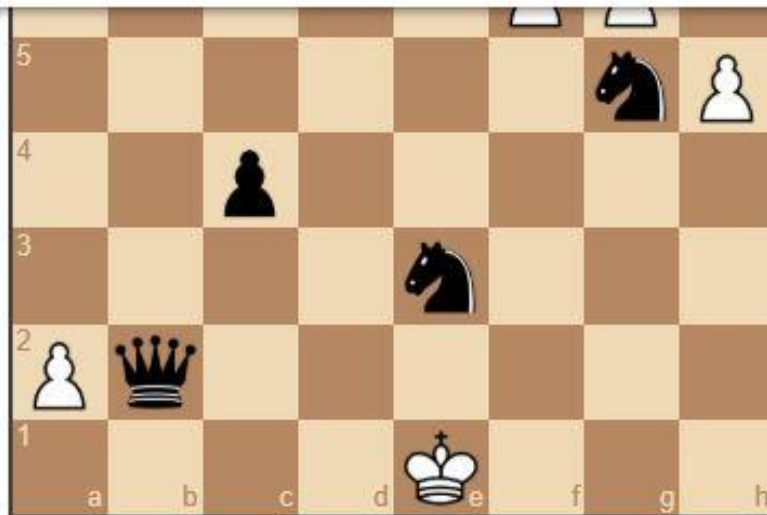


Fig 10: Intermediate Snapshot of the game

This page says

Game over

OK



Search depth: 3 ▼

Positions evaluated: 3926

Time: 0.499s

Positions/s: 7867.735470941884

gb ngsr

Ke2 Qxb2+

Ke1 Nc2+

Kd1 Ne3+

Ke1

Fig 11 : White loses the Game

8. Conclusion

This project was completed in fulfillment of the course requirement of ADA, Algorithm Design and Analysis. It provided a very good opportunity to obtain both the theoretical knowledge and practical experience. It helped us to get more familiar with the development flow of gaming systems. The result of our project meet both the requirement of the course and our expectation. A competitive game between human user and the machine was realized. This project began as fun experiment to see if we could develop a chess program that would make legal, albeit uninformed moves. It needs to have the functionality to best, or at least put up a strong fight against us in a game of chess.

In addition the development of the core program, we experimented with some simple evaluation functions and the initial results seemed to confirm what had been covered in the introductory Minimax Algorithm while considerations like safety and mobility can affect machine choices, material value and search depth tend to hold the most weight in such decisions. Perhaps the largest future extension we would propose is the addition of opening and end game databases in order to make it perform more competitively in these areas. It features a user-friendly interface.

This project provides, to the best of our knowledge, the first methodology of automatic learning the parameters of the evaluation function.

References

<https://inst.eecs.berkeley.edu/~cs162/sp07/Nachos/chess.shtml>

<https://www.cmpe.boun.edu.tr/~gungort/undergraduateprojects/Developing%20an%20Adaptive%20Chess%20Program.pdf>

<https://www.scribd.com/document/168302488/Chess-game-implementation-in-Java>

http://www2.eng.cam.ac.uk/~tpl/chess/David_Hammond/Dissertation.html

<https://en.wikipedia.org/wiki/Backtracking>

<https://en.wikipedia.org/wiki/minimax>

<https://en.wikipedia.org/wiki/HTML>