IJCSE International Journal of Computer Sciences and Engineering Open Access
Research Paper Vol.-6, Issue-7, July 2018 E-ISSN: 2347-2693

A.C.P. (ARTIFICIAL CHESS PLAYER), June 2017

Shalabh Agarwal[1*], Tapadeep Chakraborty[2], Nikita Dutta[3], Shenelle Alphonso[4]

[1,2,3,4] Department of Computer Science, St. Xavier's College (Autonomous), Kolkata, India

**Corresponding Author: Shalabh Agarwal.*

Available online at: www.ijcseonline.org

Accepted: 13/Jul/2018, Published: 31/Jul/2018

Abstract – Artificial Chess Player is a chess engine which runs on a Java platform. A 'chess engine' refers to a machine which has the ability to play the game of chess against a human subject or another chess engine. The basic functionalities of a chess engine constitute - accepting a move from its opponent, computing the most optimal move within a reasonable degree of approximation and communicate the output back to the opponent and repeat the process with the ultimate goal of winning the game. Out of these, the most significant and also the most challenging part in the construction of a chess engine is enabling it to compute approximately the "best" move to play from a given position of the game and that will be the topic of this paper. The main objective of this paper is to understand the algorithmic rules which help to guide the engine to victory in a limited resource system and also to implement the rules using a high-level language (in this case, Java). The paper will also go through the useful techniques used to implement a chess engine like minimax, alpha beta pruning, board evaluation and understand how to implement them in coding language. Although, chess engine is not a new word in the field of computer science and artificial intelligence, but it is still a field of active research even to this day when machine can beat man at his own game.
Prerequisites – A thorough understanding of the rules of chess, basic understanding of game trees, Java semantics

Keywords – chess engine, move set, minimax, Alpha-Beta pruning, board evaluating functions, zero-sum game, move ordering, horizon effect, Stockfish.

I. INTRODUCTION

A chess engine maybe formally defined in the following way – a machine/program that takes as input a valid board and outputs one among the possible set of legal moves. It may therefore be viewed as a black box mapping a board to a move.

$$F(B) \rightarrow M$$

This almost oversimplified way of looking at a chess engine is actually vital as it enables us to give a mathematical form to the complicated problem. A perfect chess engine is actually infeasible because of the really big branching factor (30, on average) of a chess game tree. Knowing that an average game lasts for 80 ply, we can compute the number of boards to be generated at 30^{80} which is about 10^{120} [1]. In comparison, the number of atoms in the observable universe is about 10^{80}. Hence we would need to limit the game tree depth to a finitely small number and search all possible configurations within that limit.

For the program to work, therefore, it must be able to generate all possible set of moves from a given board configuration (and choose one of them). Degree(s) of freedom of movement of the different pieces are defined by the following set of rules:

Table 1: Degree of freedom of each piece

Piece	*Domain of movement*
King	(x±i,y±j) where i,j=1,0,-1 and not both i,j=0
Pawn	Black : (x+1,y) White : (x-1,y)
Bishop	(x-i,y-i) (x-i,y+i) (x+i,y-i) (x+i,y+i) where 1≤i≤8
Rook	(x+i,y) (x-i,y) (x,y+i) (x,y-i) where 1≤i≤8
Knight	(x±i,y±j) where i,j=1,2 and i≠j
Queen	Domain(Rook) U Domain(Bishop)

All of the above moves must additionally satisfy:

- Boundary condition of the 8x8 board
- Except for knight, no other piece is allowed to skip pieces
- King must not be put to check

```
public void progress()
{
        move_set(root);//depth 1 of game tree
        Board re=root.left;
        while(re!=null)//increase depth as per required
        {
                move_set(re);
                Board rez=re.left;
                while(rez!=null)
                {
                        move_set(rez);
                        /*Board rez1=rez.left;
                        while(rez1!=null)
                        {
                                move_set(rez1);
                                rez1=rez1.right;
                        }*/
                        rez=rez.right;
                }
                re=re.right;
        }       }
```

This game tree must be stored in a specific data structure and investigated for an output. To help us to implement the idea we shall create a class called *Board* with all information pertaining to a board and a class called *Tree* which will have a tree rooted at a node and each node in the tree will be an object of the board class.

Rest of the paper is organized as follows, Section I contains introduction of a chess engine and representing the game's essential components in the form of logical models which can be worked upon using high level programming languages, Section II contains the essential data structures required to store the data in order to operate on the Game Tree, Section III contains various techniques and functions used to evaluate a board's relative winning position, Section IV contains various algorithmic techniques used to evaluate the Game Tree and generate an output, Section V articulates the various results obtained with the final prototype of the chess engine and discusses its efficiency, Section VI concludes the paper by discussing its efficiency and the future improvements.

II. Data Structures to store the Game Tree

Each object of the Board class is itself a node of the game tree. The Board class must be equipped with the following member variables:

Apart from these, there are special moves which are allowed in a game of chess which include – pawn capture, double pawn move, pawn promotion, en-passant, castling which must also be considered with suitable condition subject to constraints. Function *move_set()* generates all possible moves from a given configuration.

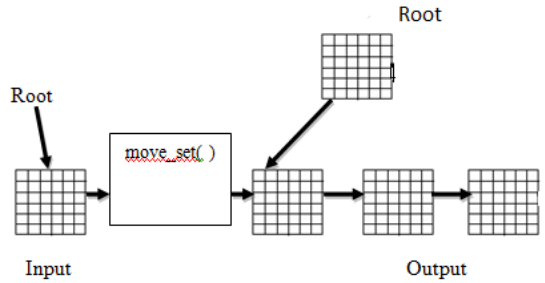

Figure 1: Black Box representation of the working of move_set function

Hence we are able to generate all possible set of moves from a given board setup. All of these "child-boards" maybe similarly expanded using the same rules, and repeating this process a number of times for each board up to a specific depth generates the game tree. This process can be done using a function called *progress()*:

- *turn* – equal to 1 indicates white to make a move and -1 indicates black to make a move
- *depth* – indicates the depth of the board in the current game tree
- *kings_rooks_moved* – one requires prior knowledge of whether the four rooks and the two kings have been moved, to realize a castling move. One may use separate variables or an array to store the Boolean values.
- *enpassant_condition* –information must be stored with respect to en-passant as it can be availed for just one ply
- *Board left and right* – to store links to left (child) and right(sibling) boards in the game tree
- *2D board array* – to store the board configuration with specific number/symbols indicating specific pieces
- *board_value* – stores a double real number indicating who is leading the game and by what margin

Pieces in the 2D array storing the game tree maybe represented using distinct numbers, while piece color maybe identified using signed representation of the same number. In the project in hand, the following have been chosen as values of various pieces:
|Rook|=2, |Knight|=3, |Bishop|=4, |Queen|=6, |King|=5, |Pawn|=1
So a typical starting board will look like:

-2 -3 -4 -6 -5 -4 -3
-2
-1 -1 -1 -1 -1 -1 -1
-1
0 0 0 0 0 0 0
0
0 0 0 0 0 0 0
0
0 0 0 0 0 0 0
0
0 0 0 0 0 0 0
0
1 1 1 1 1 1 1
1
2 3 4 6 5 4 3
2

With Black pieces occupying rows 0 and 1 and White pieces occupying rows 6 and 7.

In order to properly implement the game tree one must have a reliable data structure which can be traversed efficiently and saves memory. Although we may use a simple tree structure we choose not to do so. This is primary tree structure cannot be used because the branching factor of a game tree is not constant (it may range from 0 to more than 100). Hence we would need to implement a data structure which may work for any number of child nodes.

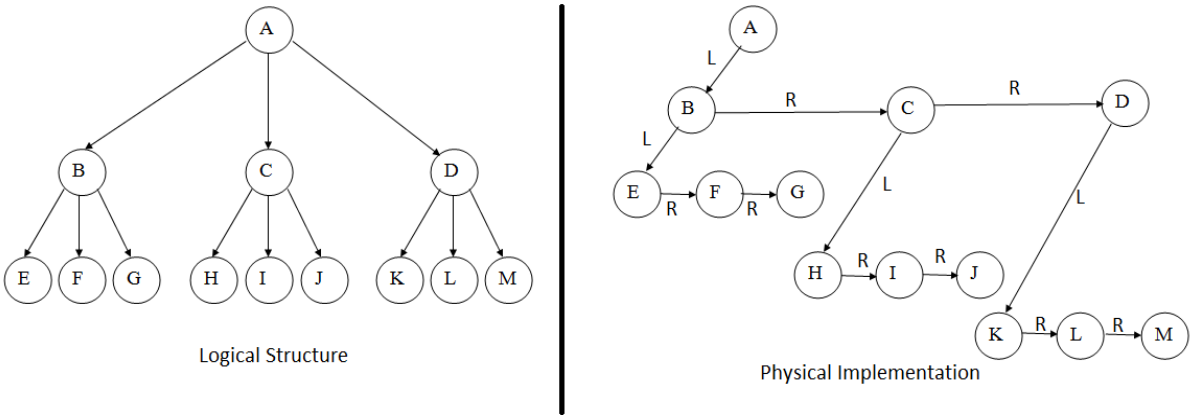

Figure 2: The Data Structure

The Game tree rooted at A to the left maybe represented physically in the memory as the tree to the right. To implement this tree we require two distinct kinds of links – left (L) and right(R). If a node X in the tree is L-reachable from Y then X is the child of Y, i.e. to reach node X from Y, we need to pass through one left link. Two left links indicate grandparent-grandchild node relation and so on. Whereas nodes that are reachable using R links and no L links are at same level and sibling nodes. In the above diagram nodes E, F and G are all the possible board configurations possible from node B. Few advantages of designing the tree in the above fashion are:

- Can work for variable branching factors
- A simple preorder search achieves dfs (depth first search) searching of the entire tree

The tree rooted at the present game position maybe expanded to a desired depth depending on memory constraints. The entire tree may have thousands of nodes depending on the depth to which the tree is limited. However the following simple recursive code will display the entire tree:

```
public void explore_tree(Board rx)//displays the tree in dfs fashion
{
        if(rx==null)
                return;
        rx.display_board();
        System.out.println("\n");
        explore_tree(rx.left);
        explore_tree(rx.right);      }
```


III. Board Evaluating Functions

Before discussing board evaluating functions, one is required to understand what is meant by *end-game* situations. A certain valid board configuration is in end-game if any one of the following hold true:

- It is white to move and white has no legal moves available and white king is in check
- It is black to move and black has no legal moves available and black king is in check
- It is white to move and white has no legal moves available and white king is not in check
- It is black to move and black has no legal moves available and black king is not in check

The first and second condition is checkmate with black and white winning respectively and the last two indicate the condition for stalemate. We can say that the first condition is the worst configuration for black and hence we may assign the value -∞ (worst for white) to it and for similar reasoning assign ∞ to any board which satisfies the second condition. For first stalemate, we assign a very low value for white (but not as low as being check-mated) and also penalize black for not being able to checkmate and losing mobility and vice-versa for the forth condition.

Now when we expand the tree to a certain depth (say k), we are required to search for the best move available to us within the given scope of visibility but we make an obvious observation that not all leaf nodes of the game tree till depth k, necessarily satisfy any of the end-game conditions. Hence it is essential to devise some kind of metric to measure who (black or white) is leading (which chess players do intuitively). Since chess is a two player *zero-sum* game, one party leading necessarily implies that the other party is trailing. We therefore use the following metric system to compute how likely a board condition is favorable towards the white player:

$$f(B) = \sum_{i=1}^{b} WiPi$$

The function takes as input a board and outputs a real number. In other words, the function maps a board to the set of real numbers. A positive value would indicate white leading the game and negative would indicate black leading (or, white trailing). The formula is a linear combination of feature(s) with weights. It is essential to realize that the weights may depend on several factors and in all conditions may not be static. For instance, a rook which is forked/blocked by neighboring pieces is not as valuable as a bishop which has more mobility. However a standard maybe adopted as a reference, the weights which were used by A.C.P. were:

Table 2: Weight multiplier standards

Piece	*Weight multiplier*
White Queen-Black Queen	900.00
White Rook-Black Rook	500.00
White Bishop-Black Bishop	400.00
White Knight-Black Knight	350.00
White Pawn-Black Pawn	100.00

[Note: The above values are not absolute and were chosen after experimentation, under specific conditions the values maybe altered for better evaluation]. These weights must be multiplied with the number of piece difference. For example, if white has two rooks and black has none we add 2*500 to the board score of *f(B)*. Using the above form of evaluating function will make a sound but weak chess engine because likelihood of victory doesn't depend merely on the number of pieces one player is ahead at, but also depends on several other dynamic factors including mobility, safety of the king and so on. These ideas are required to be materialized and incorporated in the chess engine as bonus factor or penalizing factor. Some of the heuristics which may be used include:

- *Tapered Evaluation* – This heuristic basically suggests the use of different weights in different game situations of opening game, mid-game, closing game.
- *Bishop Pair Bonus* –Having the bishops side by side allows the player to control a lot of boxes enabling better control of the game. So we may give a small bonus if such a configuration is ever achieved.
- *King Safety* – The king must always be protected and must not be prone to being checked easily as it drastically reduces move choices and hence mobility. The king must also not be surrounded by too many same colored pieces as it might block potential escape routes. Having an early castling have been observed to be an advantage in terms of safety.
- *Pawn Development* – This is perhaps the most essential factor in the game of chess. An underdeveloped pawn structure is a sign of vulnerability as it indicates that the player is controlling less number of blocks and also pawns must be developed as it may be promoted to major pieces on reaching the final row. Thus a pawn maybe systematically given a higher weightage maybe given to pawns which are ahead than pawns which are relatively backwards. The weight scales for white pawns which were used in A.C.P. were 110, 120, 130,

200, 400 for each row the pawns progress. Pawns in the penultimate row maybe considered as valuable as a bishop because it is just one step away from being promoted. Hence, a pawn in the penultimate row is a potential queen

- *Pawn Clusters* – When pawns are side by side they can support each other and but when pawns are isolated or doubled (one before another), they end up blocking the progress. Hence some amount of negative (with respect to white) must be given to penalize such configurations.
- *Mobility* – Mobility is also a very important aspect which essentially says that having more options open at each step ensures an added advantage. So the more legal moves a player has in his possession, the better. Mobility scores may also be calculated for specific high priority pieces like the queen.

On top of all the above mentioned factors, many more maybe introduced for better performance of the chess engine.

[Note: Only leaf nodes in the tree has to be evaluated for the computation]

IV. Algorithmic Techniques to evaluate the Game Tree

MINIMAX

After successful generation of the game tree, one has to implement algorithms to find the best move in the game based on the limited scope of visibility (depending on the depth of the tree). One of the most obvious approaches would be implement some form of backtracking. The algorithm which uses this is termed minimax. Minimax assumes that the opponent is at least as intelligent as the chess engine. Since we assign the board values relative to white, so black tries to attain a score as low as possible whereas white tries to attain a score as high as possible. In other words, white acts as a *maximizer* and black acts a *minimizer*. Starting from the bottom level, the algorithm chooses the minimum or the maximum of the possible choice of values, depending on the parent node being maximizer or minimizer, and repeats the same process in a bottom-up fashion. The process has been demonstrated in the following illustration using a simple binary tree and arbitrary node values:

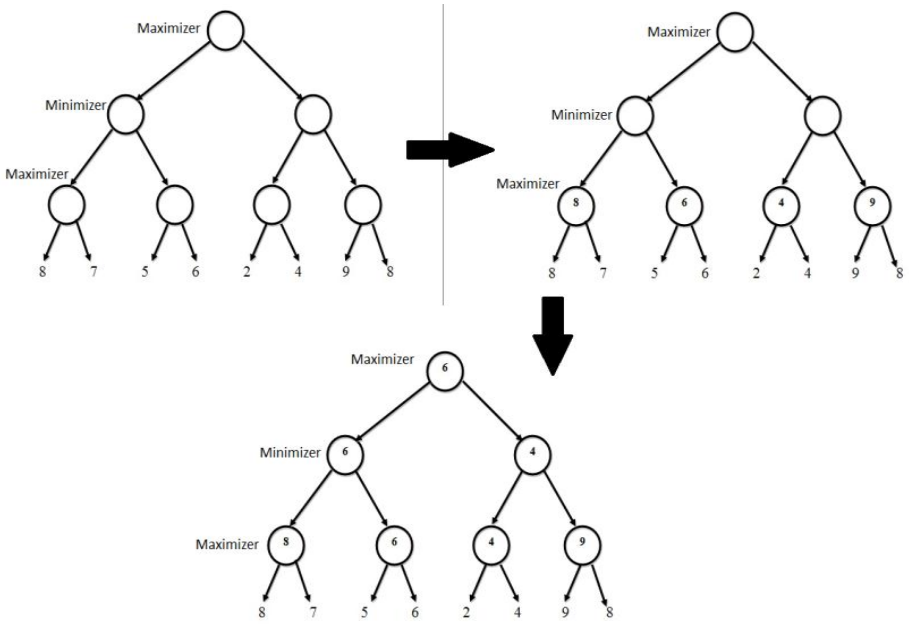

Figure 3: MINIMAX algorithm

In the above example, the best move for the player at root would be to go left but one may be initially persuaded to move right because the highest value lies to the right sub-tree of the root but as we can see that he might also end up getting a value of 4, if he were to move right.

ALPHA-BETA PRUNING

Alpha-beta pruning is not a new algorithm but an improvement on the existing minimax. This technique increases the efficiency of minimax significantly. It stops completely 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. The algorithm maintains two values, alpha and beta, which represents the minimum score that the maximizing player is guaranteed and the maximum score that the minimizing player, is guaranteed respectively. Initially alpha is negative infinity and beta is positive infinity, i.e. both players start with their worst possible score. Whenever the maximum score that the minimizing player is assured of becomes less than the minimum score that the maximizing player is assured of (i.e. beta ≤ alpha), the maximizing player need not consider the descendants of this node as they will never be reached in actual play. The illustration below shows alpha-beta pruning in action:

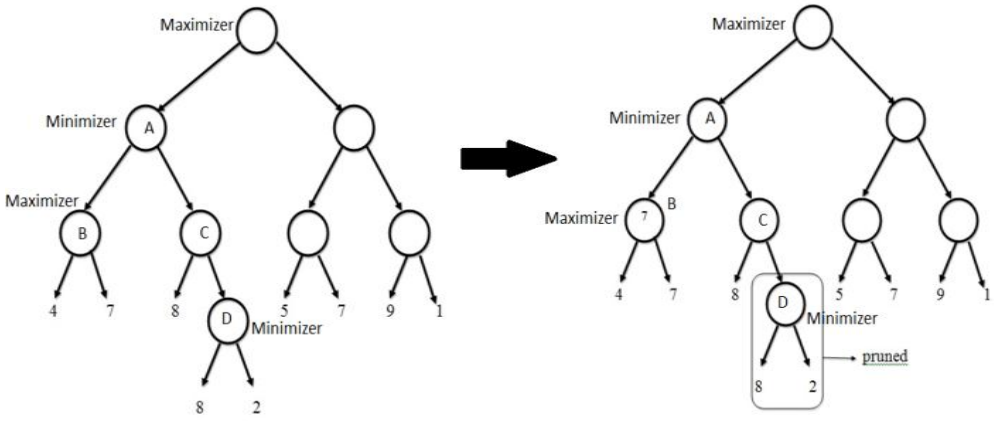

Figure 4: Alpha-Beta Pruning

Unlike minimax which makes a decision based on its immediate parent, alpha-beta pruning makes its decision based on the grandparent node. In the above example, when the minimizer A gets a value of 7 from its left sub-tree and observes a value of 8 (greater than 7) in left sub-tree of C, it decides not to investigate C's right sub-tree D (which maybe substantially long). This is because with respect to A, 7 is the best value he can get, irrespective of the value obtained by node D. Minimax works best when the nodes are sorted in descending order of their board values, this is called *move ordering*.

The code which can be used to implement minimax with alpha beta pruning using the modified tree is shown below:

```
public double minimax(Board r)
{
        if(r.left==null)
                return r.board_value;
        Board p=r.left;
        double f=minimax(p);
        p=p.right;
        while(p!=null)
        {
                double q=minimax(p);
                if(p.turn==1)//minimizer
                {
                        if(q<r.board_value)
                        {
                                r.board_value=q;
                                return q;
                        }
                        if(f>q)
                                f=q;
                }
                else
                {
                        if(q>r.board_value)
                        {
                                r.board_value=q;
                                return q;
                        }
                        if(f<q)//maximizer
                                f=q;
                }
                p=p.right;
        }
        r.board_value=f;
        if(r.right!=null && r.right.left!=null)
                r.right.board_value=r.board_value;
        return f;
}
```

With minimax and alpha beta pruning in place, the entire algorithm maybe graphically represented using a control flow flowchart. The flowchart incorporates all the major operations and in which order they need to be repeated in order to obtain a working model of a chess engine which can keep playing a game of chess until an end-game position is reached. The flowchart is illustrated below:

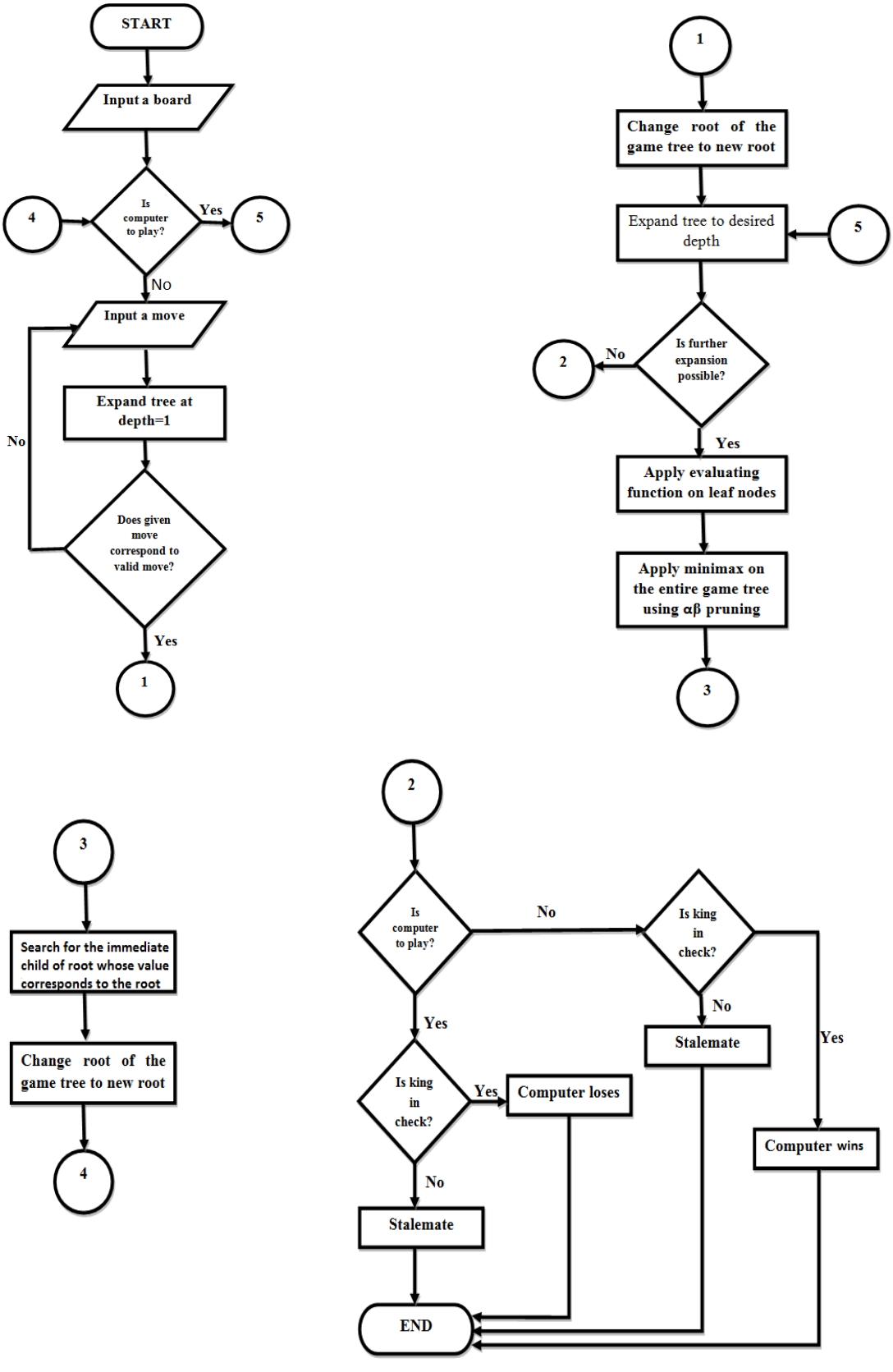

Figure 5: Flow chart of A.C.P.

Certain features like *Autoplay* was introduced in the final project where the algorithm can basically keep making moves being the white and the black player alternatively. This feature maybe used to test the strength of various evaluating functions where white uses a certain set of functions and black uses a different set. Other useful features like *Retrieve Move* was introduced to give players additional flexibility to retrace a move if they want to change the move they had previously played – this can be achieved using a linked list storing the various board configurations in the course of the game. Finally there must be validation checks to ensure the moves given by the user are legal moves abiding by the rules of chess.

V. RESULTS AND DISCUSSIONS

Few of the results of the final prototype are discussed below. The developed chess engine was tested using *lichess* chess simulator and was also tested against the most popular and one of the most powerful chess engines namely *Stockfish*.

In the first phase of testing of the project the obvious errors and vulnerabilities were detected and removed. These tests included ensuring proper castling conditions are maintained, valid pawn promotions are implemented and ensuring en-passant move is played with correct constraints in place and so on. Any bugs that were found were rectified and a fully functional chess engine was developed. These tests are not mentioned in here with the assumption that the reader is fully aware of the rules of the game.

The next phase of testing included testing strategic weaknesses which the chess engine might suffer from and which from a developer's perspective should be removed. In the graphics provided below, the right side of the illustration shows the actual moves being inputted to the machine in *Command Prompt* window and left side shows a simulation of the actual chess board using *lichess*. Few of the cases are discussed below:

- *Checkmate:* Although checkmate opportunity can be easily spotted by a human subject and put to effect immediately, a machine maybe fooled and never actually play the final move but always be in contention to checkmate the opponent. The following example illustrates the problem and shows a graphic of how it should behave after the problem is solved

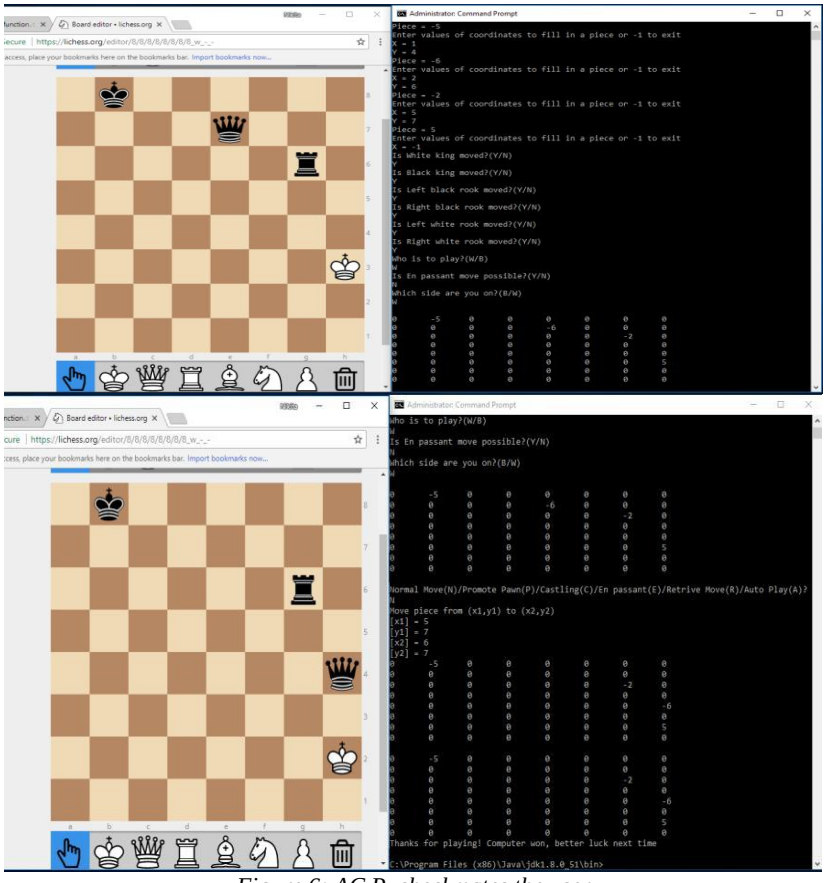

Figure 6: AC.P. checkmates the user

- In this game, the computer is playing black. This is an essentially an end game situation as there is no way for white to stop checkmate. However we observe that there are various ways to checkmate white from the initial position, the engine treats all checkmates equally so a checkmate after two moves is good as a checkmate in the immediate next move. This problem means that the chess engine can repeatedly inspect the game tree and keep being in a dominated position and never actually make the checkmate move. To eliminate this problem, the *board_value* is multiplied by a factor of *1/depth* which means a checkmate in the node of lesser depth is preferred to a checkmate further down the game tree.
- *Queen Threat:* Queen is an essential major piece in the game. It is perhaps the most important piece after king. Running a queen down in a game of chess is considered a fatal position. Hence the queen must be steered to safe positions and steered out of difficult positions. The slides below show exactly how the engine responds when the queen is put to threat. The computer is playing black in the illustration below. In this game, the black queen in coordinate d7 is in danger as the white rook in coordinate d5 will capture the queen in the next turn. If the queen captures the rook in its move, the white pawn in coordinate e4 guarding the rook will capture the queen in turn. The engine, in order to save its queen, moves the queen from coordinate d7 to coordinate a4. Hence, the algorithm successfully withstands a queen threat and steers the queen to the safest spot, in turn putting the knight in harm's way.

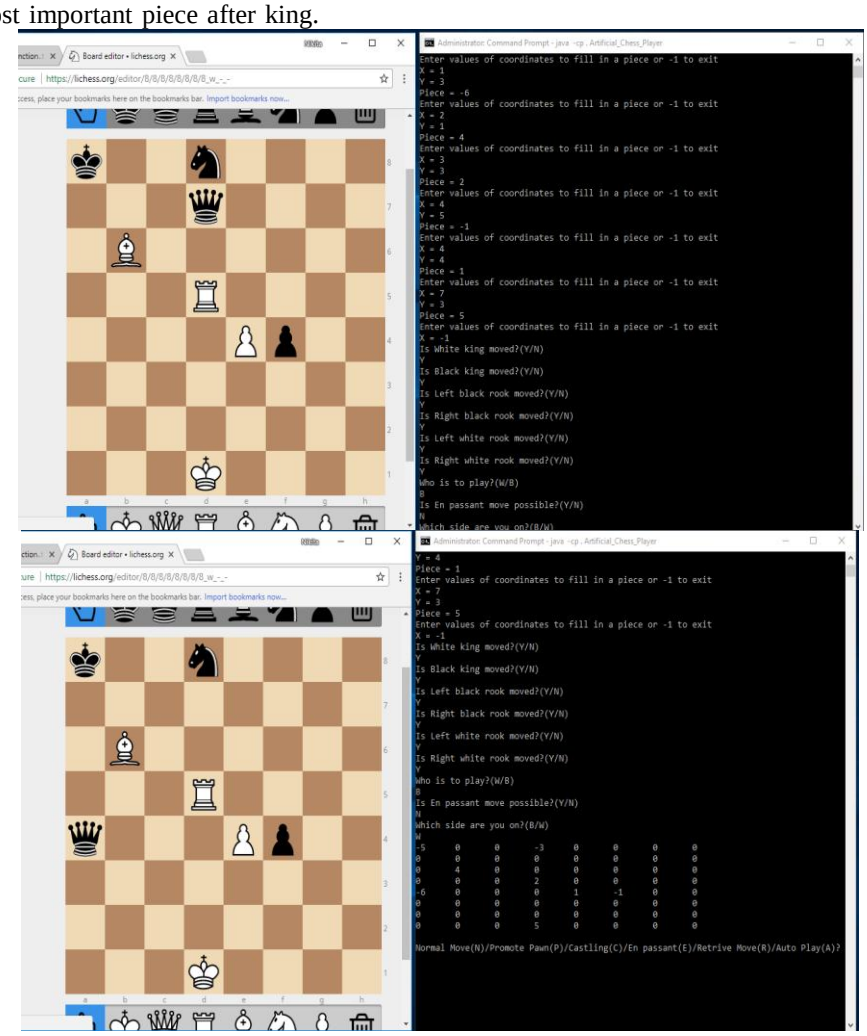

Figure 7: A.C.P. saves the queen

- *Fork:* Forking is a very common technique used in chess to capture a piece and/or to limit the opponent player's options. Forking happens when two pieces are challenged simultaneously by a single opponent's piece, forcing the player to sacrifice one for the other. Usually forking is the strongest challenge when the king is put to check and another major piece is also under attack, usually this means that the player has to move the king and sacrifice the other major piece. A forking example is shown below, where computer is playing white and black has a forking

option next move and it is white to move. The black knight at a4 can fork the white king and the white queen by moving to c5 meaning that white will have to exchange a queen for a knight. The chess engine is also given an immediate option to capture the black rook at c8 with white bishop at e6 as a bait to see if it takes up the immediate bait for foresees the incoming fork. The chess engine indeed realizes the threat and moves its queen from d3 to d6.

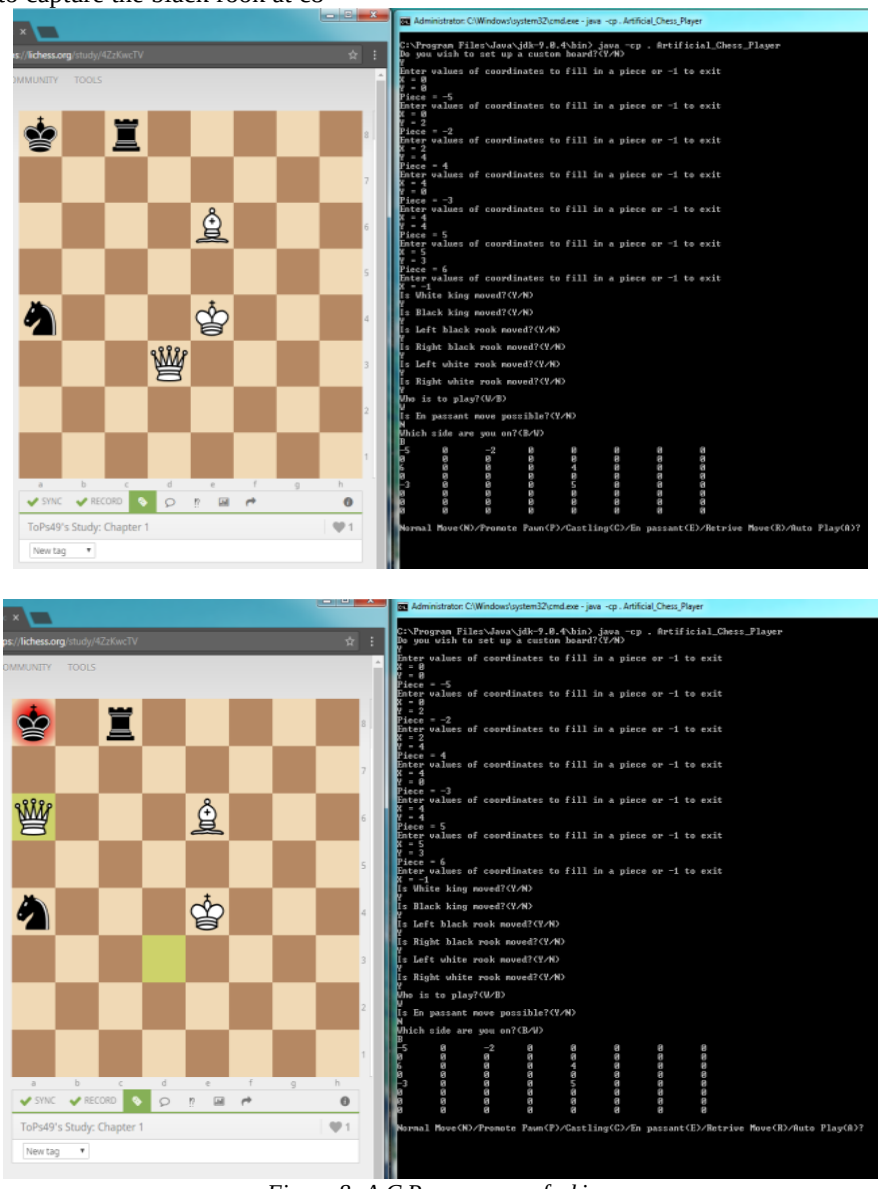

Figure 8: A.C.P. overcomes forking

VI. CONCLUSION

Using alpha-beta pruning along with minimax with strong board evaluating functions, one can design a chess-engine which is reasonably fast and reasonably efficient. Because of the backtracking problem, minimax has a runtime complexity of $O(b^d)$, where b is the branching factor of the game tree and d is the depth of the game tree. With the introduction of alpha-beta pruning the efficiency increases to $O(b^{d/2})$ at best case with perfect move ordering [2]. The fact that makes such a complicated game is because it belongs to *EXP*-class of problems. There exist more games of chess than there are atoms in the observable universe. Due to these reasons chess has always been a challenge for both man and machine. The only theoretical problem this chess engine suffers from is the *horizon effect*, where the chess engine simply overlooks certain obvious game traits because the game tree is shallow and the problem cannot be removed simply by increasing the depth of the game tree, because the horizon may be present anywhere.

ACKNOWLEDGEMENT

Artificial Chess Player was guided by Prof. Shalabh Agarwal without whose unconditional guidance and constant support this project would not be a success. His patience, motivation and immense knowledge were essential for the completion of this project. The professors of the Department of Computer Science of St. Xavier's College are also to be credited for this thesis especially Dr. Anal Acharya for providing his valuable time and expertise. Last but not the least, Dean of Science, Principal and Vice Principal of St. Xavier's College, Kolkata earns gratitude for allowing us to do this.

REFERENCES

[1] Claude E. Shannon, "Programming a Computer for Playing Chess", Philosophical Magazine, Ser.7, Vol. 41, No. 314 - March 1950

[2] D. Klyushin, K. Kruchinin, "Advanced Search Using Alpha-Beta Pruning, Matematychni Studii. V.25, No.1

Author's Profile

Shalabh Agarwal is an Associate Professor in the Department of Computer Science, St. Xavier's College, Kolkata. He is also the Director, Computer Centre & Central Computing Facilities and Head of the Department of Computer Science at the College. His research Interests are Green Computing, Pervasive Computing, Software Engineering, IOT and Internet security.

Tapadeep Chakraborty graduated with Bachelor in Computer Science from St. Xavier's College (Autonomous), Kolkata. He is presently pursuing M.Sc. in the field of Scientific Computing from Savitribai Phule Pune University and his interests include complexity theory, machine learning and other aspects of theoretical Computer Science.

Nikita Dutta is graduated with Bachelor in Computer Science from St. Xavier's College (Autonomous), Kolkata. She is currently doing her M.C.A. from Vellore Institute of Technology. She is interested in android app development, mobile computing, cloud computing and study of various programming languages.

Shenelle Alphonso graduated with Bachelor in Computer Science from St. Xavier's College (Autonomous), Kolkata. She is presently working with Ernst & Young as a professional in Cyber Security. Her interests include cyber security, soft computing and business analytics.

