CHESS ENGINE USING BITBOARDS

VARUN MUTHANNA
Graduate Student
Computer Science Department
University of Texas at Dallas
vkm150030@utdallas.edu

Abstract

The purpose of the project is to develop a chess engine and analyze various techniques that help in reducing the computation time. Bit-boards are used for board representation and move generation, which provides considerable improvement compared to representing the board using arrays. Algorithms like depth first search, minimax game tree, alpha-beta pruning are implemented for quick search of the best possible move up-to any given depth. Evaluation of the board will be done based on the material balance, material position and material mobility to help the search. The chess engine will have the support for UCI(Universal Chess Interface) protocol so that it can be integrated with any available Graphical User Interface(Arena).

1. Introduction

1.1 Motivation

Chess is described as Drosophila of artificial intelligence[1], in the sense that it has spawned lot of research in AI much like the study of common fruit fly towards the discoveries in genetics and many of the AI techniques can be implemented and tried in chess program. Chess is a controlled environment where the computer is presented with the situations and goal, and it is expected to find the goal using various decision making algorithms. Hence it would provide great understanding on AI techniques during the study and implementation of the chess engine.

1.2 Technical Challenges

Chess is defined as a game of “perfect information” as there is no secrecy and the state of the game is known to both the players all the time. The goal is to draw strategies to capture the opposition King by capturing as many opposition pieces without losing much of our own. To achieve this, following are the requirements for the computer to plan the strategies.

a) *Board Representation* : Some way to represent the board so that state of the game is known all the time.
b) *Move Generation* : Generate all legal moves without breaking any rule of chess.
c) *Choose best move* : Once all the moves have been generated the engine should make the decision to choose the best one. The problem one would face is, a good move at one level might turn out to be bad in subsequent level. Hence search to many subsequent level is required.
d) *Evaluation* : Numerical rating is required for every possible move thus enabling the engine to easily compare the different available moves and choose the best move.
e) *Integration* : In order to test the engine, the user needs a easy interface to play against the engine. So we have to support for UCI protocol in the engine and hence can be integrated to any chess GUI.

1.3 Approach

The technical challenges will be addressed using the following approaches :

a) *Board Representation* : Bit-boards are used to represent the board. Each square is considered as a bit in 64-bit word. This is efficient as most of the present systems are 64-bit, and bit operations are computationally less intensive.

b) *Move generation* is one of the complicated aspect and computationally intensive as we need to take care of all the rules of chess. Use of bit-boards in board representation helps in big way to reduce the complexity and computation time.

c) *Choosing best move* : The minimax algorithm is used where white tries to maximize the board value and black tries to minimize the value. Since we are required to search deeper(depth of search is called "ply") to get the best possible move we face huge computation requirement. Hence we use alpha-beta pruning to cut down unnecessary search paths.

d) *Evaluation* : Evaluation is considered to be one of the most important aspects for the chess engine to perform well. Proper evaluation functions will help in finding better moves even at small depth search. In this engine Material, position and mobility valuations will be used.

2. Bit-Board Representation

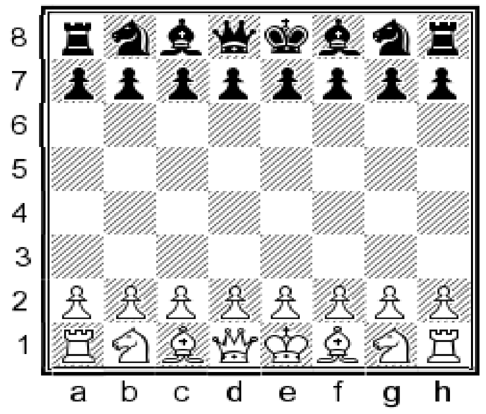

Fig 1: Chess board with labels

In bit-board representation we use 64 bit integer to represent the board(long in Java). So each square is represented as a bit in the 64 bit word. In a chess board we have 8 rows(called Ranks represented as 1,2,3,4,5,6,7,8), and 8 columns(called Files represented as a,b,c,d,e,f,g,h) and each of these ranks and files can be represented as 64bit integer so we have Ranks[8](Ranks[0] for 1st rank and Ranks[7] for 8th rank), Files[8](Files[0] for a file and Files[7] for h file) representing corresponding ranks and files. The presence of a piece is marked as 1 else it is 0. Then how do we know which piece is present in which square? For this we need few more 64 bit integers.

We initialize a 64 bit integer for each type of piece of respective color. Hence we need 12 64-bit word, one each for white king(WK), white queen(WQ), white rook(WR), white Bishop(WB), white Knight(WN), white pawn(WP), black king(BK), black queen(BQ), black rook(BR), black bishop(BB), black knight(BN), black pawn(BP).

So any state of the board can be obtained by performing logical OR on all these bit-boards

```
        long OCCUPIED = WK|WQ|WR|WB|WN|WP|BK|BQ|BR|BB|BN|BP;
        long VACANT = ~(OCCUPIED);
```

Thus we can obtain the board with just logical operators instead of looping through 64 times to check each square.

3. Move Generation

Based on the rules of movement of the pieces they can be divided into two kinds :

1) Sliding pieces(Rook, Bishop, Queen).
2) Non-Sliding pieces(King, Knight, Pawn)

The move generation procedure is similar for all the pieces in particular group. Now let us see how these techniques will reduce the time complexity.

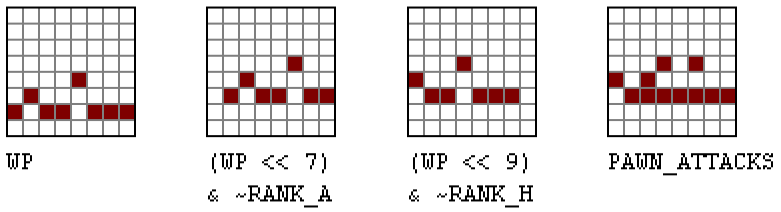

Fig 2: pawn attacks for white (from O'Reilly bitsets using java)

3.1. Non-Sliding Pieces

Pawn :

All the moves for all the pawn of a particular color can be obtained by just four shift operation on the pawn bit-board. Fig-2 shows how to obtain the attacking squares of the white pawn. The pawn can move one position or two position(from beginning position). This can be obtained by

```
         White_Pawn_moves = (WP << 8 ) | ((WP <<16) & Ranks[3]));
```

If we had not used bit-boards we had to loop through 64 times to check the position of the pawn and generate corresponding moves and attack squares.

Move generation for other non-sliding pieces like Knight and King can be obtained in the same way with simple left and right shifts to all its legal positions.

3.2. Sliding Pieces :

Sliding Pieces are of two kinds diagonal sliding pieces(Bishop) and horizontal-vertical sliding pieces(Rook). Queen is both diagonal and horizontal-vertical sliding piece.

To see how to generate the possible moves for sliding piece let us take an example of one rank with the presence of Rook in one of the squares.

p	p				R		p

Fig 3 : Single Rank in a chess board

Now we need find all the positions the rook can move. Notice if there is a piece in between then the rook can capture that piece but cannot have moves beyond that.
Now lets represent one rank shown above in bits which is 8 bit word. But while we are considering the whole board we will take all 64 bits.

OCCUPIED = O = 11000101   (board bitmap)
slider = S = 00000100   (rook bitmap)
O – 2*S = 10111101
left_attack = O ^ (O – 2*S) = 01111000 (thus we got the left positions rook can move)
we can similarly obtain the right attack using same as that of left but by reversing it.
Let O' represent reverse of O
so right_attack = (O' ^ (O' -2*S'))' = 00000011
Therefore total horizontal rook moves at that rank is left_attack | right_attack = 01111011

Now replicating this to whole chess board, we can obtain the OCCUPIED(O) by performing bitwise AND with the precomputed mask for each rank(Rank[8]) which is 1 on that Rank and 0 everywhere else. For vertical attack we have precomputed mask for each File(File[8]) to obtain OCCUPIED(O).

Similarly for finding the moves of the diagonal sliding pieces we use precomputed left diagonal and right diagonal masks using rotated bitmaps and thus obtain the all the diagonal moves for diagonal sliding pieces.

Using the methods as discussed above we can arrive at all the moves with just few bitwise operations and the precomputed masks. Thus reducing the computation time drastically.

4. Search Algorithms

Now since we have generated all the moves it is time to search for a good move. A intelligent program should be able look at the board and determine who is ahead(With the help of evaluation function), and devise a plan to get the best advantage to itself. Here the White will try to maximize the board evaluation and Black will try to minimize it during their turn of play. This is called the **MiniMax** algorithm.

We will try to find recursively for every move what the opponent would do and find a best move for us considering even the opponent plays the best move. This will be done up-to certain depth, which will help the engine to either find a trap or devise one for opposition.. Hence we have a game tree to search before making every move. But we have a problem here. In the opening game, each side will have 20 choices and will increase up-to 35 in middle games which will give us a huge tree to search and will take lot of time even for small depths. So we use **alpha-beta pruning** to remove necessary search paths which give huge reduction in computation time.

To see how alpha-beta pruning works let us take an example :
Suppose we have already searched for max, path through move A is giving evaluation 20. Then we move towards B. The min at B has two path C and D. when we search C we get the evaluation 0. At this point we know no matter what the value at path D, min will always choose value less than or equal to 0. hence we need not search path D and max can directly choose path A as it would give better evaluation.

To achieve MiniMax and alpha-beta pruning in the single tree search, a depth-first search(DFS) is required. As DFS needs a stack data-structure to be implemented, and recursion functions use stack inherently, so we can use it to simplify our implementation.
Below you can find a simple algorithm I have used :
We initialize the alpha and beta value to a high value considering it to be infinity. Here we are taking alpha = -20000 and beta = 20000 which comparatively infinity for the values returned by evaluation function.

```
function minimax_alpabeta(alpha, beta, board, depth,player)
    if depth == max_depth
        return evaluate_board(board);

    if player == White
        allmoves = get_all_whitemoves(board);
    else
        allmoves = get_all_blackmoves(board);

    for each move of allmoves
        makemove(move);
        score = minimax_alpabeta(alpha, beta, board, depth+1,player);
        if player == Black
            if score < beta
                beta = score;
        else
            if score > alpha
                alpha = score;
        if alpha >= beta                    //Alpha-Beta pruning in done here
            if player == Black
                return beta;
            else
                return alpha;

    if player == Black
        return beta;
    else
        return alpha
```

5. Evaluation Function

Evaluating a position is considered one of the most difficult and important aspect of chess programming. The success of MiniMax search is greatly dependent on the valuation we do. There are various factors which will define the value of the board for a particular player. I am considering three most important of them.
a) Material Balance – This will give an account of which pieces are on board for each side.
b) Positional value for each piece – This will give the positional advantage that a player has on other.
c) Mobility – The greater the mobility of pieces a player has, more advantage he holds.

To support the concept of MiniMax where white tries to maximize the board evaluation and Black tries to minimize the board evaluation, we will take positive value of white valuation and negative value of the Black valuation done using above three methods.

5.1 Material Balance :

Each piece in chess have different values and combined values of some pieces will be less than some individual piece. So, care should be taken while valuing the pieces. Taking into the above features I have the following values for each piece(Hans Berliner's system).

Pawn = 100, Knight = 320, Bishop = 333, Rook = 510, Queen = 880.

Care should be taken so we don't lose the King so it should be assigned a very high value and I have given King =5000.

5.2 Positional Value :

To achieve the goal of winning the game(capture the opponent King) the pieces should control the board. This can happen only if the pieces are positioned in proper squares. Hence we increase or decrease the value of the pieces based on the position they are present at. We use Piece-Square table to evaluate the positional value. Here are the Piece-Square table of each white piece.

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

PAWN

-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

KNIGHT

-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

BISHOP

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

ROOK

-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

QUEEN

-30	-40	-40	-50	-50	-40	-40	-30
-30	-40	-40	-50	-50	-40	-40	-30
-30	-40	-40	-50	-50	-40	-40	-30
-30	-40	-40	-50	-50	-40	-40	-30
-20	-30	-30	-40	-40	-30	-30	-20
-10	-20	-20	-20	-20	-20	-20	-10
20	20	0	0	0	0	20	20
20	30	10	0	0	10	30	20

KING

Positional valuation will help the engine in lot of aspects which are important in controlling the board and move towards the goal faster like
a) Pawns will be developed early and will not block other pieces.
b) Minor pieces like bishop and knight will be bought to center early so it can take control of center and attack the opposition pieces
c) Rooks are freed up so they can move freely
d) Queen will stay at the center and keep threatening the opposition pieces through out the board.
e) King will not get into opposition territory and hence end up being captured.

5.3 Mobility :

The greater the number of moves available the greater the player can threaten the opposition. So each move available is given 5 points so as to make pieces not block each other.

Other valuations are available like changing material and positional value based on beginning or middle or end games, increase value if 2 bishops are available, reduction for double pawns and isolated pawn, reduction for moves to unprotected squares and so on. But in this engine I have just taken three simple and most important valuations.

6. Experiments and Results

As mentioned before the alpha-beta pruning will reduce huge amount of computation time. We will evaluate the performance improvement obtained by alpha-beta pruning by performing the search with and without alpha-beta pruning at different depths(2, 4 and 6) for initial position(beginning game) and after 15 moves(can be considered as middle game) for a random chess game played.

The table below shows the time taken for the search which is an average value of the four attempts performed on each condition.

2-Ply search

Time taken in milliseconds	Without Alpha-Beta	With Alpha-Beta
Initial game	72 ms	38 ms
Middle game	102 ms	48 ms

4-Ply search

Time taken in milliseconds	Without Alpha-Beta	With Alpha-Beta
Initial game	425 ms	198 ms
Middle game	1426 ms	280 ms

6-Ply search

Time taken in milliseconds	Without Alpha-Beta	With Alpha-Beta
Initial game	111636 ms	1736 ms
Middle game	1298724 ms	8345 ms

From the numbers we can notice that we are gaining huge time in the search. For example at 6-ply search for the move in the middle game the engine takes a mere 8 seconds with the help of alpha-beta pruning which otherwise would have taken 20 minutes.

7. Conclusion

In this paper we have clearly seen the advantage of using the bit-boards in the board representation and the move generation where the time complexity of O(n) obtained by representing the board using the array can be reduced to O(1) (constant time). Also by the results obtained by the experiment on the search algorithm gives us an estimate on the improvement obtained by using alpha-beta pruning and how a simple concept will help the chess engine choose the best move in short time.

As of the development of the Engine software, it is complete and has been integrated with the UCI protocol. It was tested using Arena open-source GUI. With 4-ply deep search the computer is able to beat a middle level chess player in a 5 minute blitz game.

References

[1] Nathan Ensmenger Is chess the drosophila of Artificial Intelligence? A social history of an algorithm.
[2] Robert Hyatt Chess board representation https://www.cis.uab.edu/hyatt/boardrep.html
[3] Robert Hyatt Rotated bitmaps, a twist on an old idea https://www.cis.uab.edu/hyatt/bitmaps.html
[4] O'Reilly on Java.com Bitwise optimization in java : Bitfields, Bitboards, and beyond http://archive.oreilly.com/pub/a/onjava/2005/02/02/bitsets.html?page=2
[5] Chess programming blog by François Dominic Laramée http://www.gamedev.net/page/resources/_/technical/artificial-intelligence/chess-programming-part-i-getting-started-r1014
[6] Blog on Chess AI http://www.aihorizon.com/chessai/intro.htm
[7] WIKI of chess programming https://chessprogramming.wikispaces.com/
[8] Logic crazy by Jonathan Warkentin https://sites.google.com/site/jonathanwarkentinlogiccrazy/
[9] Full documentation on UCI protocol http://wbec-ridderkerk.nl/html/UCIProtocol.html
[10] Notes on Minimax with Alpha Beta pruning http://web.cs.ucla.edu/~rosen/161/notes/alphabeta.html
[11] Chess evaluation function http://chessprogramming.wikispaces.com/Simplified+evaluation+function
[12] Blog on chess programming http://mediocrechess.sourceforge.net/index.html