Overview of a Simple Chess Implementation

by Joey K Black

Introduction

Several weeks ago I started looking for a new topic to study. I wanted something really interesting that would keep me challenged for a long time. In college I studied graphics, but I have no real artistic talent, so that was a bit of a dead end. The past year or two I've studied web applications, but I never really came up with anything interesting to create. Finally I have arrived on the topic of artificial intelligence.

AI is a topic with almost infinite depth. There are several major fields and allot of work still to be done. In the next several years it is possible that computers will finally be comparable in power to the human brain. If software developers can keep up with hardware advancements, we could be seeing real (strong) AI in the next few years. This my be fairly optimistic, but I think its still fairly more realistic than the wildly optimistic views of the mid 60s.

Chess

For my first AI project, I decided to tackle the classic challenge of teaching the computer to play chess. I have started off simple. My AI uses a simple minimax implementation that is only capable of looking a few moves ahead. For the data structure, I used bitboards. These simple structures are a really efficient and compact way of representing the board.

Tutorial

In order to implement a chess program, you will need to understand a few basic concepts. First, you need to know how to play chess. I will not cover that here. In this brief tutorial I will cover bitboards, bitboard operations, minimax algorithm and leaf evaluation. This should be enough to get you started.

Bitboard

A bit board is a primitive variable who's binary representation is used to represent the location of a certain type of piece on the board. A 0 represents an empty location and a 1 represents an occupied one. In chess you have 12 types of pieces and 64 squares on the board. So, to represent an entire chess board you need 12 64 bit values. In java that means 12 longs.

You can map you bits to squares however you like, but I mapped it so the lowest bit (the 1's position bit) is the bottom left square (A1) and the highest bit (the sign bit) is the top right square (H8).

Bitboard Operations

The advantage of a bit board is not just compactness, but also the ability to use bitwise operations to move pieces. For example a pawn can be moved with the following operation:

pawnBitboard << 8

This will move every 1 bit in the bit board up one row. To attack you could do:

(pawnBitboard << 7) | (pawnBitboard << 9)

That is basically all there is too it. For an actual implementation, you will need to take into account wrap around (the pawn on a2 attacks h3), but this can be solved by ANDing the result of pawnBitboard << 7 with a bit board representing all the positions that are not in file h:

((pawnBitboard << 7) & ~*FILE_H*) | ((pawnBitboard << 9) & ~*FILE_A*)

Pieces that slide are a bit trickier. They can only slide until they hit another piece. These pieces can be moved with a few extra bitwise operations. The method for implementing a rook is as follows.

First, you will calculate one direction at a time (up, down, left, right). Do this by shifting your piece 8 times and then ANDing the bitboard with the rank or file the piece is on (to prevent overflow). Example for moving a rook to the right:

rightBoard = ((board << 1) | (board << 2) | (board << 3) | (board << 4) | (board << 5) | (board << 6) | (board << 7)) & rank

Next you need to negate out all the occupied positions. Notice I shift the the enemy occupied positions by 1 or more while I include the unshifted friendly occupied positions. This is so that I am including captures of enemy pieces:

long rightOccupiedFriendly = (rightBoard & friendly);
long rightOccupiedEnemy = (rightBoard & enemy);
long rightOccupied = (rightOccupiedFriendly | (rightOccupiedFriendly << 1) | (rightOccupiedFriendly << 2) | (rightOccupiedFriendly << 3) | (rightOccupiedFriendly << 4) | (rightOccupiedFriendly << 5) | (rightOccupiedFriendly << 6)) | ((rightOccupiedEnemy << 1) | (rightOccupiedEnemy << 2) | (rightOccupiedEnemy << 3) | (rightOccupiedEnemy

<< 4) | (rightOccupiedEnemy << 5) | (rightOccupiedEnemy << 6)) & rank;
rightBoard &= ~rightOccupied;

Once you have done these two steps for every direction, you just need to OR the 4 directions together and you have your move map. For the queen, you just OR the move maps for a rook and bishop together.

Now what about black? Instead of reimplementing all these operations for black, all you have to do is flip your bitboard, perform the calculations as if it was for white and then flip the board back. Flipping the board is fairly simple:

```
(RANK_8 & value) >>> 56 |
(RANK_7 & value) >>> 40 |
(RANK_6 & value) >>> 24 |
(RANK_5 & value) >>> 8 |
(RANK_4 & value) << 8 |
(RANK_3 & value) << 24 |
(RANK_2 & value) << 40 |
(RANK_1 & value) << 56
```

Other useful calculations can be done using bit boards. For example, suppose you want to know how many pieces you have in the center of the board. First you create a bitboard representing the center squares then you AND it with all your pieces. Or suppose you want to test if you are in check. Just OR all the other players pieces' moves together then AND it with your kings bitboard.

More advanced operations can be done using things like rotations and magics, but this is enough to get a simple, but effective, chess program up and running.

Minimax Algorithm

The minimax algorithm is based on a fairly simple assumption: For every move you make, your opponent will choose the counter move that is best for him. So when calculating the value of making a move, you can deduct from that value the gain your opponent will get from countering that move 'perfectly'. By doing this calculation, you can MINImize the MAXimum gain of your opponent.

I implemented the minimax algorithm by looping through all possible moves and choosing the one with the highest value. I calculate the value by adding in any intrinsic value of making that move (ex. capturing a piece) and subtracting the value of the opponent's best move. The opponent's best move is calculated recursively using the same method.

Of course you cannot let the recursion continue indefinitely. The number of moves that have to be searched increases exponentially as you look further ahead. I used a simple depth counter to determine when I should stop the recursive calls.

Leaf Evaluation

Once you reach the maximum depth on your minimax search, you still have to determine a value for the moves you are on. Since this move search represents a tree of possible moves, we call these the leaf moves.

For the leaves, you can first calculate the value of any captures that move makes. For example, you could award 100 points for capturing a pawn, 300 for a night, etc. You can also add in a value if the other player is in check, if you have pieces in a certain area of the board (pieces in the center are worth more), having rooks on an open file, and so on.

Next Step

Once you have a program that can store piece positions, generate valid moves and search through valid moves to select optimal moves, your computer is essentially ready to play chess. But that dose not mean it will play very well.

From here, numerous improvements can be made. I used allot of objects to wrap my bit board and move representations. This could be eliminated to speed things up. I did not expect my chess program to be winning any competitions, so I used the object structures to simplify things.

More advanced algorithms could also be used, such as Alpha-beta pruning. Alpha-beta pruning discards moves in that are provably worse moves than other moves that have already been analyzed, thereby reducing the number of moves that must be searched.

An opening library and an end game database can also provide a great improvement. At the start of the game, chess programs can only see so far ahead and they lack the knowledge of an experienced player who knows how to start off right. A library of tried and true openings can fix this. The same is true for the end game. There are databases of end game positions that have been analyzed in advance that can be used by a chess program to determine a long series of moves that can force a checkmate. Without the pre-calculated database, the chess program often can not see far ahead enough to force a win.

Finally, my implementation could be improved by porting it to a faster platform. C++

would give some improvement. Programming it to use the GPU would improve it even more. The fastest platform of course, would be to build the computer using specialized chess hardware.

Reflections on Bitboards and Intuition

One of the major advantages of a human player is their ability to 'feel' whether a position is good or not. For example, an experienced player can intuitively tell if a position is too exposed. Without really looking too far ahead, they can tell just by the current state of the board.

How do they do it? Pattern recognition. The human brain recognizes patterns by compressing sets of similar patterns to a simple pattern that identifies these similarities. For example, we recognize a human face as a face because it matches our overarching pattern of an generic face. We can then remember an individual face by simply storing the the things that uniquely distinguish this face from the generic one. We even seem to derive our concept of beauty from the degree to which a face deviates from the generic face. The more a face matches our personal compressed pattern for a face, the more 'ideal' it is.

So in chess, a player who has seen numerous positions and seen how those positions played out has subconsciously compressed groups of positions into good and bad patterns. A chess program could theoretically do the same thing. It could learn intuition by keeping track of positions and how they play out and using some sort of compression algorithm to create patterns that seem to be good or bad.

At the expense of rambling (a price that has probably already been paid), I would also like to reflect on the apparent meaning that begins to emerge as data is compressed. As the program learns and compresses groups of similar positions, they begin to have meaning. A simple number (used as a bitboard) is now 'good' or 'bad' based on what has been compressed into it. This is similar to the way humans associate meaning. We compress our experiences into patterns and this leads us to associate these patterns to words thereby giving the words meaning. You can only understand what it means to be 'happy' or what makes you 'happy' if you have had a set of 'good' experiences to compress into a pattern you can associate with that word. As this concept is explored, many concepts previously considered mysterious and uniquely human, begin to be explainable and thereby theoretically programmable into an AI.

Conclusion

Well, I think I covered all the major topics needed to implement a simple chess program, as well as some additional thoughts of my own. Chess AIs, and AIs in general, have come a long way, but they still have a long way to go. We 'ugly bags of mostly water' still have a few advantages. How long these advantages will last against the impending robot invasion, only time will tell.