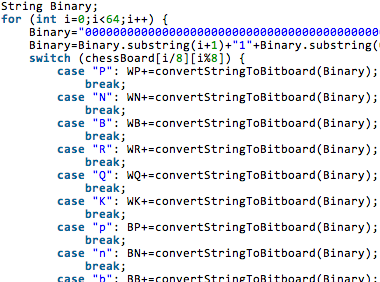
**Complete Java Chess Engine**

**Simple Games: Man vs. Machine**

**Ryan Tibbetts**

**Legacy**

The initial intent of this project was to produce a multi-application board game AI. The AI could read many games as long as their rules and parameters were defined. This AI was achieved, but the programs applications were incomplete and not presentable in a final product. Under the time constraints I made an executive decision to not finish the applications and instead turn in a finish product of my initial work for the application of chess. Before I even began the multi application AI I thought it important to develop the most difficult application for the AI to read, Chess. I was successful in this, but the completed software used incompatible syntax for working with other applications of similar parameters so I had to start over. This happened 3 weeks prior to now. The end result was a much more compatible package with a design very similar to my previous chess engine, but the work proved too tedious and here we are now.



[A portion of code from the multi-application project. Note the use of binary. This was much more preferred. Arrays worked equally well, but processes slower.]

In this research paper I will cover a relevant history behind the game as well as the software’s interface and the AI. This includes an explanation of the games requirements, rules implementation, logic, and search algorithm.

**Introduction**

Some time prior to 6th century AD, game was made that seemed to simulate a battlefield and possibly even the complexities of strategy within war. Similarly, it was often told that some used it in place of needless war and settled their differences over this intelligent game. It has been passed down for 2 millennia and to this day is exceedingly well known to all. The game of which I speak is of course chess.

Around the year of 1770, Charles Babbage, a renowned mathematician and engineer, suggested the bizarre idea of a mechanical chess player. His original design for the first mechanical computer was, believe or not, somewhat key in the initial development behind the predecessor of modern day chess[4]. He did not however make an engine to play chess. Had he access to such technology we have today, maybe he’d have been the first.  
 About 180 years later in 1950, two renowned mathematicians of the 20th century, Claude Shannon and Alan Turing, competitively produced the first complete outlines for a chess program[5]. This development was apparently all that was needed to get dreams to happen, because 9 years later someone produced the very first fully functioning 8 by 8 board game program of Checkers! Although not chess, this made great leaps in the programming community for the future of a chess engine. 

[Claude Elwood Shannon]

Around 1970, chess playing computers were something to be had by the public[1]. A great development from where it all started, however not the final goal. Beating the common man was not so difficult with a computer that could procedurally generate moves and predict faster and further than you. What about the pro’s? It wasn’t until 1996 that a match took player where a computer by the name of Deep Blue beat the world champion chess player[7].

I’m not here to make Deep Blue, however I did decide that producing a formidable AI to be proper, but what syntax would I use? I ultimately decided on Java for the sake of ease in creating the GUI. This did however make the program slower than if I had chosen perhaps C++, and honestly after all said and done I might have liked myself better by the end as well. Java is well known for its finicky interface merely because a lot of it is predesigned and in all honesty I ran into minor difficulties due to this. Regardless of code I needed a structure to my program, so lets talk about the requirements.

**Requirements**

Present day there are many fine chess engines operating at an elo’s far above 2000, so as you can imagine the process to make an engine is probably fairly well know. This was not the case so I found and I had to start from scratch. The first thing I needed was a set of rules backing up the parameters for which I would code:

1. Chess is a game with two persons who take turns in making their moves.
2. Chess is a game where both players know the current state of the game (Nothing is hidden between the players)

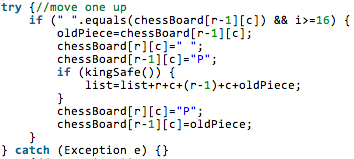
These two parameters allowed me to have some insight into the initial development of my Chess Engine. They severed even more use when it dawned on me the methods for which to develop the game. To create something an AI could read, I would need rules.

**The Rules**

I assume you the reader know at least the basic moves within chess so I will skip the specific piece by piece rules of the game and review the general concepts that the AI would need to comprehend.

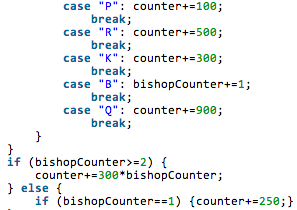
1. Turns
   * The AI would need to know that it would not play the game by itself, but rather recognize the predictable state of an opponent so that it may use this knowledge in a search engine.
2. Moves
   * The AI would need to know the moves for each piece as well as the legalities for their moves.
3. Value
   * The AI would need to understand the value of a piece as well as the value behind its position on a board.
4. Objective
   * The AI would need an objective that could hopefully be defined by all prior mentioned concepts.

**Turns, Moves, Value and Objectives**

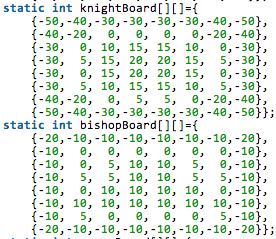
Not so unexpected, the aforementioned concepts were very synergetic in overall construction of this software. I figure it best to cover them all together since they interact so closely. Lets start with the moves. 

[Portion of pawn moving code]

To implement the moves, I needed to clearly define parameters for which to make moves. This was conveniently as simple as saying a pawn could move 2 spaces on the first move and could En Passant the very next had a piece moved to either side of it. Likewise it could move one space forward on the first turn, and could make an attack moving towards the opposing side and diagonally at any opportune time. Similarly the move methods would be able to recognize it’s surrounding within its move radius to confirm a proper move. The way this was done was by simply using an incriminator in a “for loop” that contained the matrix of the board as well as the position of every piece on the board.

Each pieces code followed suite, but this meant nothing if a piece would make a move that to a human made no sense. The program could clearly see that the move was possible, so why not make the move? This is where value came into play. I needed a rating system for each piece so I could define what moves the AI should be making. Each piece was given a simple rating to define its value. I gave each piece a real value between 1 and 10 so that the computer would know where the value was in making a move. This was however naïve and ineffective. 

[Porton of chess piece rate code]

I learned very quickly that these values did not stop the computer from making dumb moves. Simply put the computer would ignore most other objectives if it meant it seeing itself with more points. This lead to the AI’s inevitable failure and it was easily tricked by most human players. There was more depth needed to the concepts behind move making. 

[Portion of board positional rate code]

The solution was to make a board for each piece type. These boards would be the same matrix as the field, but lacked the pieces on it and in their stead contained number values. Each square within the board had a different value defining the value of a position on the board such to add and subtract the pieces current existing value. A few relative value changes later I was able to design what I think was an effective board to piece interaction to help define what was and wasn’t a good move to make for each piece.

The objective was very simple with all of the above in mind; get the most the most points! Right? It wasn’t as simple as that. The computer will not predict moves past its pre-coded limit, however it can take an awful long time to compute all those possible trees when at just 4 depths it would have to search exactly 197,281 nodes. The solution was a pruning method.

**Algorithm**

If (depth search == 0 or the total possible moves == 0)

{Return the new piece values for the game and turn}

Search for unlikely moves and ignore their tree such as to shorten the search.

Determine player’s side and begin algorithm**;**

For (The length of the full list of possible moves after pruning has been done)

Computer makes a move**;**

Flips board in order to not have to know more move methods than that of one side**;**

Computer then proceeds to view all values for x depth by recursion**;**

Depending on the players predicted moves either alpha or beta will be chosen for a select/pruned node created by the process of recursion along with a sort method for node hierarchy within the search tree**;**

If (It is the computers turn and the value of the alpha node is greater than or equal to the beta node)

{The beta node move is returned}

Else

{The alpha node is returned}

If (It is the computers turn and the value of the alpha node is greater than or equal to the beta node)

{The beta node move is returned}

Else

{The alpha node is returned}

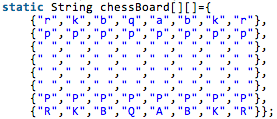
This is done recursively until the best move is chosen from selective pruning and a tedious recursive process that develops numerous nodes to be analyzed.

**Additional Work**

As per request, I have added the following sections to my research paper to discuss my multi-application research: “Additional Work,” “Single vs Multi,” “Preft Search.” Considering both single-application and multi-application work similar, the creation of my multi-application was much easier after finishing the single-application software.

**Single vs Multi**

Much like the single-application AI, the multi would need to be able to comprehend my prior list of rules. There was virtually no difference in the way the AI was to work for the applications, however their syntax greatly varied.

Where the single app was made out of a single array containing 

[A string matrix of chess]

all chess values, the multi-app used what be call a “bit board” a single line

**:Screen Shot 2014-04-30 at 6.15.33 PM.png**of 1 and/or 0. In the case of the example above it is and empty bit board.

In the same way that my single-app used a matrix for each piece rating, the multi app used a bit board to contain each piece. For example, 00000000 is one row on the chessboard and 0000000000000000000000000000000000000000000000000000000000000000 is the whole board. So to set up the white pawn board the bit board would look like this:

0000000000000000000000000000000000000000000000001111111100000000 respectively and the black pawn like this: 0000000011111111000000000000000000000000000000000000000000000000 respectively.

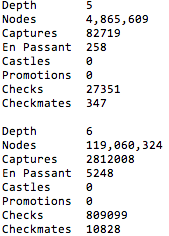
This of course meant I needed to use longs to perceive columns and rows, but that was simple enough. This was actually a bit harder to conceptualize and put together, but with help from a friendly tutorial I was able to produce and comprehend my work.

Although the difference between the two applications is just one particular feature within the interface methods this made all the difference. With a string matrix I would have to completely remake every single feature that went into the AI’s comprehensive inner workings. There was practically no code I could recycle and the AI would be written with completely different interface inputs. However, with a bit board I could simply make a new bit board for the pieces I would use in other games and give those boards defined moves sequences. It was really as simple as making new pieces.

Unfortunately this method, although faster, more efficient and all around better, had one downside, debugging. With the string matrix my debugging was limited and I could not do much about it, but with a bit board debugging was very possible and extremely tedious. I eventually worked out kinks in the code all the way to the 6th depth through what is called a perft search algorithm. By then I still had not produced a functioning AI, nor had I updated the algorithm’s syntax for the new bit board chess engine.

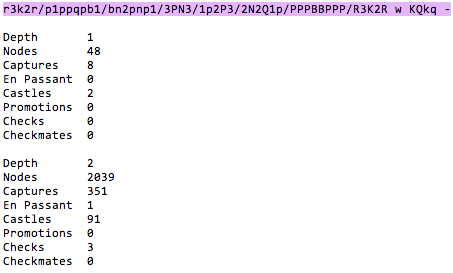
**Perft Search**

The debugging wasn’t much of a process in the single-app package, but like I said in my previous section that was not the case in the multi-app. Lets talk about the perft search method I used.

Perft search is a brute force algorithm to check every possible node that could be produced off of the first 20 moves within chess. Since the amount of possible moves and the possible variations on moves is known, we could use a prewritten data sheet containing this information to manually observe errors. Data on this sheet may look something like this: 

[A snippet from Perft Data.txt]

These values represent the amount of moves made per move type. Such should be the recorded values when using perft search and if not, I was required to determine where to little or to many moves were made and what type of moves they were. This was not the only way to utilize perft search however.

By using a chessboard syntax called FEN, I could preset the board pieces and have the perft search obtain all possible methods for which the game could end up with pieces in these exact positions. 

[FENstring preset with search depths]

There were many preset values such as this obtained in an online catalog about FENboard syntax for perft search debugging. You will find the link to this page in my Perft Data.txt within the ChessEngine package.

**Conclusion**

The Algorithm actually proved to be the easiest part out of context, however showing a computer step by step what to do did not prove easy. That being said, putting a theory to practice is just plain difficult. I now know this from first hand experience.

It is a bit disappointing that I did not actually get my single-app program to move En Passant or perform Castling, but I decided it fair that these properties were unnecessary to display the software’s ability to read and understand multiple concepts and put them into practice within a this Chess format. However, I did finish these moves in my multi-app package.

A few more bugs are in the single-app package, like lack of color change with side choosing and buggy moves at depth searches of roughly 5th depth and up. By the time I get a grade back on this I will already have added these features merely for my own personal play with the game, but for now feel free to use my code as you please and enjoy yourself some mediocre chess, unless you turn the engine level up of course.

**References**

[1] Chess.com

<http://www.chess.com/blog/billwall/the-slateatkin-program-and-chess-xx>

[2] Chessbin

<http://www.chessbin.com/>

[3] Chesspresso

<http://www.chesspresso.org/>

[4] Clark, Liat. "Turing's achievements: codebreaking, AI and the birth of computer science". Wired. Retrieved 11 November 2013.

[5] Gleick, J. (2011). The Information: A History, a Theory, a Flood. London: Fourth Estate. p. 104.

[6] Logic Crazy Chess <https://www.youtube.com/channel/UCmMjMHTeUEBJJZhxix-N-yg>

[7] Saletan, William (2007-05-11). "Chess Bump: The triumphant teamwork of humans and computers". Slate.