IMPLEMENTATION OF CHESS

A PROJECT REPORT

Submitted By

2021166978 – ARCHIT TRIVEDI

During the Course of

ARTIFICIAL INTELLIGENCE (CS 6364.002)

Under

MASTERS OF SCIENCE

In

COMPUTER SCIENCE

Erik Jonsson School of Engineering & Computer Science

University of Texas at Dallas, Richardson

Spring 2015

ABSTRACT

In this modernized world where IT industries are developing by leaps and bounds and Computers have become the basic requirements, it is indeed very important to have a strong, intelligent and reliable system. Artificial Intelligence is a field of study which studies how to create computers and computer software that are capable of intelligent behavior. This project is about Implementation of standalone application of Chess using Artificial Intelligence to improve its performance.

For the same Artificial Intelligence based algorithms are implemented. For instance Minimax algorithm and Alpha Beta Approximation are being used in order to get faster results. But this project is not only about their implementation, there are different ways in which Chess could be implemented. I have compared these ways with the one that I will be using in this project. Also to make it more interesting various functionalities in form of strategies are added in project. Strategies like selecting game modes, difficulty levels and changing strength of enemy's (here Computer itself as a player) army.

DESCRIPTION

Chess is a two-player strategy board game played on a chessboard, a checkered game board with 64 squares arranged in an eight-by-eight grid. It is one of the world's most popular games, played by millions of people worldwide in homes, parks, clubs, online by correspondence and in tournaments. In this project where in I am implementing Chess game, wherein we play with computer system as our opponent. Thus for the same Computer System must be able enough to defend itself and attack at the same time with intelligence like its opponent. In this project I will be implementing Chess using some strategies and for the same I will be using Artificial Intelligence by implementing Minimax algorithm and for further fast solution, Alpha Beta Approximation will also be used. Project implementation will be using Java. Let's see complexity of a Chess Game:

- 20 possible start moves, 20 possible replies, etc.
- 400 possible positions after 2 ply (half moves)
- 197281 positions after 4 ply
- 7 positions after 10 ply (5 White moves and 5 Black moves)
- Exponential explosion!
- Approximately 40 legal moves in a typical position
- There exists about 10^{120} possible chess games

STRATEGIES

Certain strategies will be developed in order to make it more interesting and challenging and these strategies are,

Depth value for Searching: User can give depth value to Chess engine and it will search for best move till given depth. By using higher depth value Chess Engine will be able to make best move where in vice versa will lead not lead to best move.

Branching Factor: It selects branching factor for MiniMax tree to expand. By lowering it, value tree will have lesser number of moves to consider but by increasing its value Chess Engine will have more moves to consider hence more computational time.

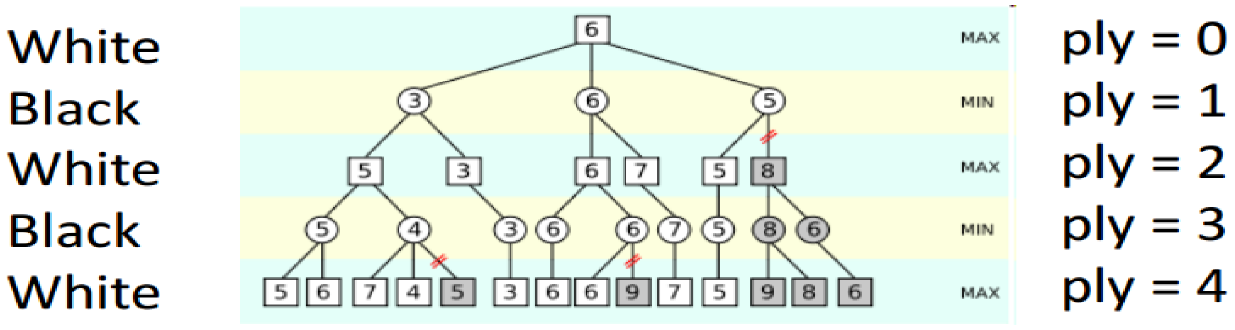
Game Mode: This engine gives 3 Game modes

- Normal: In this mode Chess Engine will behave equally for defensive moves as well as attacking moves.
- Moderately Attacking: In this mode Chess Engine will consider Attacking moves more than defensive but, if user has chosen any pieces more valuable, Chess Engine won't let them loose for attack.
- Attacking: This mode is high attacking mode. In this mode it will give highest priority to those moves which are attacking and can kill any of the pieces of opponent.

NOTE Priority for Knight, Bishop, Rook and Queen: Chess Engine gives 3 types of priorities for these pieces - Low, Medium, High. By selecting Low it won't hesitate to lose that piece. By selecting Medium It will try to defend that piece most of time but, at time when by losing that piece if Chess Engine can win the game it will consider to lose that piece. By Selecting High, Chess Engine will always defend those pieces whose priority is high.

WORKING OF MINIMAX ALGORITHM

Assume that both White and Black plays the best moves. We maximizes White's score. Perform a depth-first search and evaluate the leaf nodes. Choose child node with highest value if it is White to move. Choose child node with lowest value if it is Black to move. Branching factor is 40 in a typical chess position.



The complexity of searching d ply ahead is O(b*b*…*b) = $O(b^d)$. With a branching factor (b) of 40 it is crucial to be able to prune the search tree**.**

WORKING OF ALPHA BETA ALGORITHM

Say it is White's turn to move, and we are searching to a depth of 2 (i.e. we are considering all of White's moves, and all of Black's responses to each of those moves.) First we pick one of White's possible moves - let's call this Possible Move #1. We consider this move and every possible response to this move by black. After this analysis, we determine that the result of making Possible Move #1 is an even position. Then, we move on and consider another of White's possible moves (Possible Move #2.) When we consider the first possible counter-move by black, we discover that playing this results in black winning a Rook! In this situation, we can safely ignore all of Black's other possible responses to Possible Move #2 because we already know that Possible Move #1 is better. We really don't care *exactly* how much worse Possible Move #2 is. Maybe another possible response wins a Queen, but it doesn't matter because we know that we can achieve *at least* an even game by playing Possible Move #1. The full analysis of Possible Move #1 gave us a lower bound. We know that we can achieve at least that, so anything that is clearly worse can be ignored.

The situation becomes even more complicated, however, when we go to a search depth of 3 or greater, because now both players can make choices affecting the game tree. Now we have to maintain both a lower bound and an upper bound (called Alpha and Beta.) We maintain a lower bound because if a move is too bad we don't consider it. But we also have to maintain an upper bound because if a move at depth 3 or higher leads to a continuation that is too good, the other player won't allow it, because there was a better move higher up on the game tree that he could have played to avoid this situation. One player's lower bound is the other player's upper bound.

In form of pseudo code, alpha beta max will be,

```
int alphaBetaMax( int alpha, int beta, int depthleft ) {
   if ( depthleft == 0 )
        return evaluate();
   for ( all moves) {
      score = alphaBetaMin( alpha, beta, depthleft - 1 );
      if( score >= beta )
         return beta;   // fail hard beta-cutoff
      if( score > alpha )
         alpha = score; // alpha acts like max in MiniMax
   }
   return alpha;
}
```

In form of pseudo code, alpha beta min will be,

```
int alphaBetaMin( int alpha, int beta, int depthleft ) {
   if ( depthleft == 0 )
        return -evaluate();
   for ( all moves) {
      score = alphaBetaMax( alpha, beta, depthleft - 1 );
      if( score <= alpha )
         return alpha; // fail hard alpha-cutoff
      if( score < beta )
         beta = score; // beta acts like min in MiniMax
   }
   return beta;
}
```

DIFFERENT METHODS

Methods that can be used instead of Alpha Beta Pruning or else with Alpha Beta Pruning for betterment of Chess Programming and effectiveness are as follows:

Analyze the Best Move First: Even with alpha-beta pruning, if we always start with the worst move, we still get O(b*b*..*b) = O(b^d). If we always start with the best move (also recursive) it can be shown that complexity is O(b*1*b*1*b*1…) = O($b^{d/2}$) = O($\sqrt{b}^d$). We can double the search depth without using more resources. **Conclusion**: It is very important to try to start with the strongest moves first. But on the contrary it is not always possible to go for the best move in give search time.

Killer-Move Heuristics: Killer-move heuristics is based on the assumption that a strong move which gave a large pruning of a sub tree, might also be a strong move in other nodes in the search tree. Therefore we start with the killer moves in order to maximize search tree pruning.

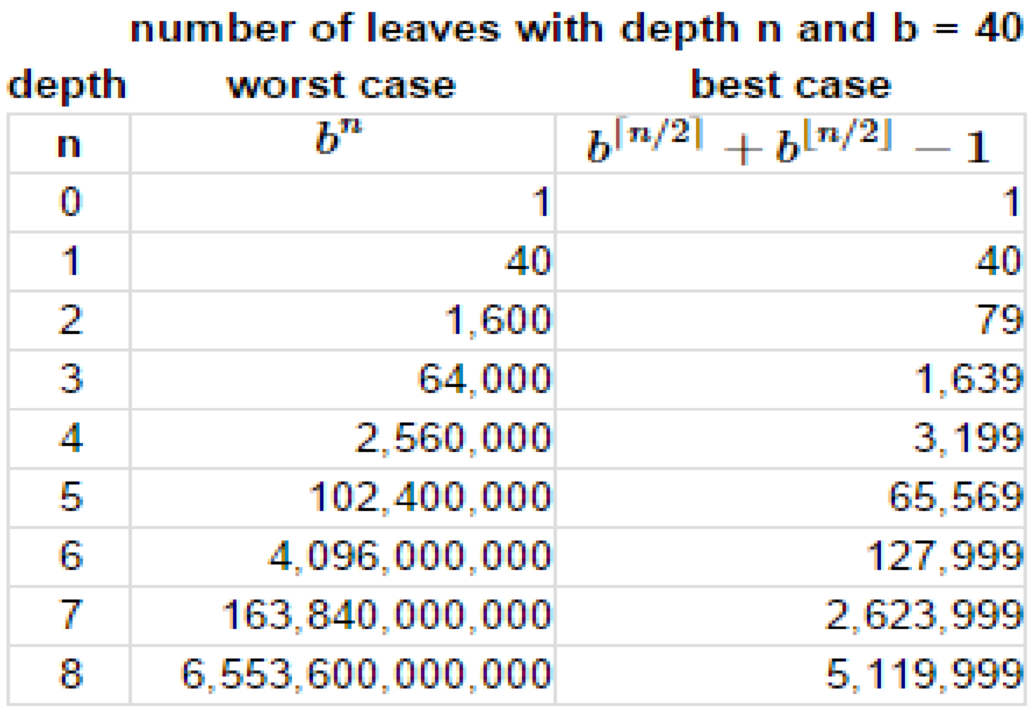
Zero-Move Heuristics: The position is so good for White (or Black) that the opponent with best play will avoid the variation resulting in that position. Zero-Move heuristics is based on the fact that in most positions it is an advantage to be the first player to move. Let the player (e.g. White) who has just made a move, play another move (two moves in a row), and perform a shallower (2-3 ply less) and therefore cheaper search from that position. If the shallower search gives a cutoff value (e.g. bad score for White), it means that most likely the search tree can be pruned at this position without performing a deeper search, since two moves in a row did not help. Very effective pruning technique.

Iterative Deeper Depth-First Search (IDDFS): Since it is so important to evaluate the best move first, it might be worthwhile to execute a shallower search first and then use the resulting alpha/beta cutoff values as start values for a deeper search. Since the majority of search nodes are on the lowest level in a balanced search tree, it is relatively cheap to do an extra shallower search.

LEARNING

The savings of alpha beta can be considerable. If a standard minimax search tree has **x** nodes, an alpha beta tree in a well-written program can have a node count close to the square-root of **x**. How many nodes you can actually cut, however, depends on how well ordered your game tree is. If you always search the best possible move first, you eliminate the most of the nodes. Of course, we don't always know what the best move is, or we wouldn't have to search in the first place. Conversely, if we always searched worse moves before the better moves, we wouldn't be able to cut any part of the tree at all! For this reason, good move ordering is very important, and is the focus of a lot of the effort of writing a good chess program. Assuming constantly **b** moves for each node visited and search depth **n**, the maximal number of leaves in alpha-beta is equivalent to minimax, $\mathbf{b}^{\mathbf{n}}$. Considering always the best move first, it is $\mathbf{b}^{\text{ceil}(n/2)}$ plus $\mathbf{b}^{\text{floor}(n/2)}$ - 1. The minimal number of leaves is shown in following table which also demonstrates the odd-even effect:

Number of leaves with depth n and b = 40



number of leaves with depth n and b = 40

depth	worst case	best case
n	b^n	$b^{\lceil n/2 \rceil} + b^{\lfloor n/2 \rfloor} - 1$
0	1	1
1	40	40
2	1,600	79
3	64,000	1,639
4	2,560,000	3,199
5	102,400,000	65,569
6	4,096,000,000	127,999
7	163,840,000,000	2,623,999
8	6,553,600,000,000	5,119,999

FUTURE WORKS:

Well if specifically I stick with Chess Programming some more strategies can be added. One such could be a Chess game developed in order to teach the user how they can play chess and work out in complex situation. One such idea was even given by my **TA Arvind Balasubramanian** about making certain plot as per the demand of user. For example a plot in which user can play with king, queen and rook of both itself and opponent. Other than this methodology should be worked out in order to deploy this code on cloud. This could be little tricky since this program would need to compile again and again causing issues in cloud framework. For example if it's coded in Java and I deploy it on Google App Engine which does not support Java then I will require a third party compiler cloud in order to compile it again and again. Further some more Artificial Intelligence and Machine Learning Algorithms must be tested in order to develop Chess.

CONCLUSION

Hereby I conclude this project report by considering Alpha Beta Approximation to be a valid way of developing Chess. It certainly reduces the work, creates optimized code. Compared to other methods Alpha Beta Approximation is a significant enhancement to the minimax search algorithm that eliminates the need to search large portions of the game tree applying a branch-and-bound technique. Remarkably, it does this without any potential of overlooking a better move. If one already has found a quite good move and search for alternatives, one refutation is enough to avoid it. No need to look for even stronger refutations.

REFERENCES

- Java Chess Engine Tutorials by Logic Crazy Chess on YouTube Channel.
- Computer Chess: Exploring Speed and Intelligence by Paul Raymond Stevens.
- Programming a Computer for Playing Chess by Claude E. Shannon.
- Articles and forum discussions on Chess Programming by wikispaces.
- Articles from Artificial Intelligence in games by Ai-Depot.
- Adam's Computer Chess Pages – Computer Chess Testing, Experimentation and Information.
- First Steps in Computer Chess Programming by Kathe and Dan Spracklen.
- Slides and materials from notes given by Prof. Dan Moldovan, UT Dallas.
- CS312 Recitation 21 Minimax Search and Alpha-Beta Pruning from Cornell University.
- One of the page from Swarthmore - http://www.cs.swarthmore.edu/~meeden/Minimax/Minimax.html.