**Chinese Chess – Algorithm Design Notes**

Board Evaluation

The evaluation function is a multivariable function that evaluates the game state by looking at the pieces and their positions on the board of both players. Using these 2 factors, the evaluation of a game board can be divided into many separate evaluations including piece *value, flexibility evaluation, positional value, cooperation/relationship evaluation* etc. Each evaluation produces a numerical value and then can be combined (not added) together to produce a single value for the board. In our game, we’ve included evaluations for piece values, board positions, flexibility value, center control, and game stage value changes. To make certain values stand out according to their importance, we sometimes used one value in the calculation of another evaluation (such as adding a piece value multiplier for the board flexibility evaluation).

Piece value evaluation is a simple calculation that looks at the number and types of piece that each player currently possesses. Each piece has a numerical value associated with it, which determines how important it is to the game. Even though the general is not a very powerful piece, it has the greatest value since the game ends if this piece is taken. The general’s value must be equal to around the value of all other pieces combined, but it must not be given infinity (Integer.MAX) or else it would override the importance of other chess pieces. Amongst other pieces, the chariot is by far the strongest due do its rulings, followed by the horse and cannon (chariots = horse + cannon approx). Some pieces can have their values increase/decrease as the game progresses (more on this later). Our ratios were taken from several online sources and then averaged. We then set the chariots value to 600 and calculated the other values accordingly. Representing piece values by hundreds allows more accuracy once we combined the piece value evaluation with the other ones. Then the piece value evaluation can be calculated by the following equation:

Pred=(*piece value*)(*number of that type of piece alive)+ (next piece value) etc…*

In general, piece value of the important pieces (rook, cannon, horse, general) contributes the most to the overall function. The computer will not make a move that gives itself a piece disadvantage even if it means getting a good position with other pieces (unless of course the computer is checkmating or gets forked by a move unforeseeable by its search depth).

Since certain pieces are more valuable at certain location based on their moving/capturing behavior. This is taken into account with a positional evaluation function. Each type of piece has its own 2-D array of integers, which the indexes reflecting the position on the chessboard. These values are important to the gameplay, as more values are given if a piece is in the correct position. For example, cannons are long-range attackers and benefit from being near its own palace, where other allied pieces easily move into place as the blocker. Contrarily, horses are short-range attackers, and obtain the most positional points for being close to the opponent’s palace. The values already factored in the piece’s own values into them, so no multipliers are needed. For example, even though a chariot can do everything a soldier can and more, the soldier gets more points for being close to the enemy’s palace since it’s worth less and can be easily captured. Therefore, the positional value is calculated for each piece of a player, and simply added into the overall function.

Even though pieces in a good position generally tend to be more flexible, this is not always the case. As such, we decided to add the flexibility evaluation. This function calculates the amount of space currently controlled by a piece and compares it with the piece’s maximum control. In our code, a piece’s control is simply defined by the number of moves it can make in the current situation. The ratio between this number and the maximum possible number of moves that the piece can make is then calculated. A multiplier is needed for this fractional value in order for it to have significance, so the piece value is used. With this, the AI is able to recognize which piece is more important to be given a greater flexibility. A secondary multiplier is used to tone down the importance of this evaluation. For example, a knight with maximum flexibility is not worth as much as 2 knights with zero flexibility. In our evaluation, I chose this value to be ¼. Finally, it is not important to give flexibility to Elephants, Advisors or Generals, since their movements are restricted. Soldiers are also excluded since they only have a maximum of 3 moves.

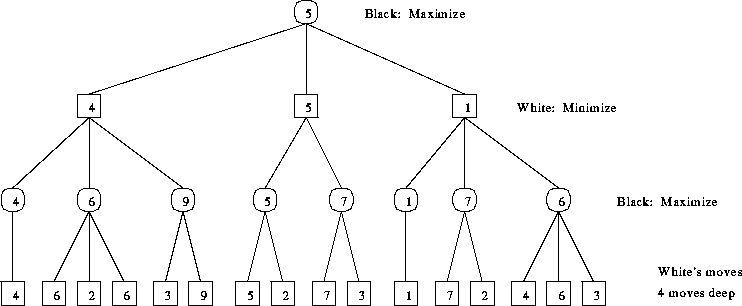
The next evaluation is the center control evaluation. This simply adds 10% of the piece’s Piece Value to the overall evaluation if that piece is inside the central region of the board. For our game, we defined a piece to be within the central region if its row position (index) is between 2 and 7, while its column position is between 2 and 6 (obtained from online sources). Even though this is an extension to the position evaluation function, it’s still important to implement since controlling the center area not only gives a positional advantage to the player, but also removes control from the opponent. Also, this evaluation has an important side effect: it allows the AI to create a strong cooperation between its pieces. For example, a horse/cannon or cannon/rook combination can create more opportunities for forking, and more pieces near the center allow a player’s cannons to gain more opportunities to attack. Finally, this value is excluded for generals, since you don’t want to use the general to control the center.

The last part of the evaluation function is not a function, but rather changes the Piece value of a piece. The game stage value change only applies for knights and cannons. As the game progresses, the cannon loses value since it has fewer pieces to set up for its attack. At the same time, knights have more flexibility as well as fewer pieces to block its movements. For our game, the Piece Values of these 2 pieces change by 1 every 3 moves made.

Minimax

To find the best possible move for our AI, we used the Minimax search algorithm. First it would helpful to understand how our tree is structured. Given a board, we generate all of the possible moves the current player can make and from those we make new board which fill the second generation. From those new boards, we would generate all of the valid moves the other player can make, adding those resulting boards to the third generation. We would continue this to any desired depth, with larger depths leading to further look ahead at the cost of longer move making time.

Each generation will represent a player, the AI being the maximizing player and the human player being the minimizing player. This is because our board evaluator will return a positive number if the current board favours the AI player and a negative value if the board is more favourable for the human player. Remember that the first generation is a single node consisting of the current board and is represented by the AI player. The AI wants to select the best move possible out of all of his valid moves. Generally, this move will lead to an advantage that was found a few moves into the future. To find this move, we must assume that both players are intelligent and that both players will choose good moves. This means that at each generation, the maximizing or minimizing player will select the most positive or negative score respectively. With that in mind, we can see how the Minimax algorithm fits well with our zero sum evaluation method. The AI will always choose the most positive score, while the human is assumed to select the most negative score.

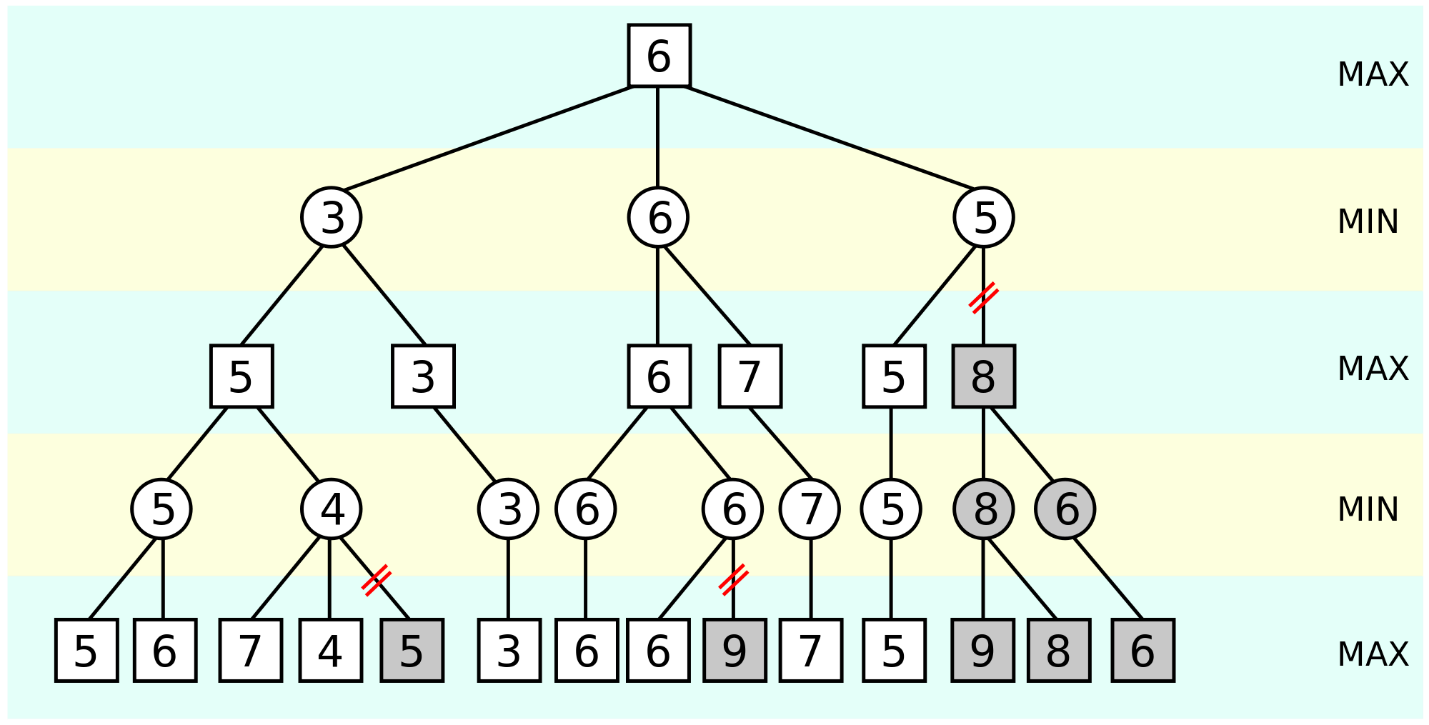


Starting from the bottom generation of the diagram above, we can see the scores of all of the non AI player’s moves. Moving up one generation, we meet the maximizing AI player who will find the highest score of the white moves and use that to score its nodes. Up another generation, the minimizing white player chooses the lowest scores of the black moves to score its own moves and finally the black player finds that his best move is down the centre branch. You may notice that from the diagram, the best move would be down the left branch to reach the board of value 9, however since we assume that the human play will not allow that to happen, the best move we can make will instead lead to a board value of 5.

To approach this, we used a depth first search using recursion. We would make two similar methods for the minimizing and maximizing player. Each method would call the other method (max calls min which calls max…) down to the desired depth or to an end game board. The method would generate all valid moves and then execute each one, one by one. After a move is executed, the value of that move would be determined by a recursive call of the opposite method. Once the recursion finishes, the moves is undone and the next move is tried. From those, the best move is kept track of and eventually returned.

Alpha Beta Pruning

To optimize the Minimax algorithm, we applied the alpha beta algorithm to eliminate portions of the search tree that do not need to be checked. We chose this method because it does not eliminate or prune off the best move. The key idea are alpha and beta which represent the minimum score that the maximizing player is guaranteed of and the maximum score that the minimizing player is guaranteed of respectively.



To see how it works, examine the diagram above. At the right, you can see that the branch starting with the 8 node is cut off. This is because its parent has already found a move with a value of 5. Its parent is the minimizing player which means that it will not choose the 8 ever. Even if that branch leads to a great score somewhere in the future, it will not matter since the minimizing player will never allow that to happen as it will chose the 5 over the 8. This means that we can eliminate that branch saving a lot of time in searching the tree. To implement this is very similar to Minimax. All we add is two variables, alpha and beta, and update their values accordingly so that early returns can be made when a branch is unfeasible to examine.

Runtime

With Minimax, we would have to search and evaluate bd boards where b is the branching factor or average number of moves possible from a board and where d is the depth to which we search. After adding alpha beta pruning, the best we can reduce this number to is bd/2 boards. However, on average it reduces the number of boards to bd-1 which is a whole generation. The math behind this is not that complex however, it will not be included to reduce the length of this document.

As you can see, the runtime is exponential which means one d approaches 5 or 6, assume b is 15, the number of boards to be evaluated will approach and surpass a million quickly. Also, because we use objects to store the board, our board evaluator will take quite some time. If we had known from the beginning, we could have stored the board using bitboards, however we feel that using objects makes the code a lot cleaner and easier to follow and debug. Come to think of it, our hardest AI searches five moves ahead which may seem little, however comparing that with a human, we realize that looking five moves into the future is actually quite a task and will actually produce a fairly strong AI.

References

Chinese Chess. (n.d.). Retrieved January 7, 2015, from https://chessprogramming.wikispaces.com/Chinese Chess

[MIT OpenCourseWare](https://www.youtube.com/channel/UCEBb1b_L6zDS3xTUrIALZOw). (n.d.). 6. Search: Games, Minimax, and Alpha-Beta [Video file]. Retrieved January 7, 2015, from <http://www.youtube.com/watch?v=6nyGCbxD848>

Li, C. (2008). Using AdaBoost to Implement Chinese Chess Evaluation Functions. Retrieved January 7, 2015, from http://statistics.ucla.edu/system/resources/BAhbBlsHOgZmSSJfMjAxMi8wNS8xNC8xNV8yNV8yOF84MzFfVXNpbmdfQWRhQm9vc3RfdG9fSW1wbGVtZW50X0NoaW5lc2VfQ2hlc3NfRXZhbHVhdGlvbl9GdW5jdGlvbnMucGRmBjoGRVQ/Using AdaBoost to Implement Chinese Chess Evaluation Functions.pdf