Final Project: Solving Chess Puzzles

Aayush Shrestha

May 2019

1 Problem Description

Our problem is to find ways to search for chess strategies to solve a chess puzzle. Chess is a two-player board game. It is played on a square board, made of 64 smaller squares, with eight squares on each side. Each player starts with sixteen pieces: eight pawns, two knights, two bishops, two rooks, a queen and a king. During the game the two opponents take turns to move one of their pieces to a different square of the board. One player (White) has pieces of a light color; the other player (Black) has pieces of a dark color. There are rules about how pieces move, and about taking the opponent's pieces off the board. The objective of the game is that each player tries to checkmate the opponent's king. Checkmate is a threat for the opponent King that no movement can stop. This will result in ending the game and player who did checkmate to another player wins.

In our case for a chess puzzle, the board setup of different pieces is given, and our algorithm has to search for moves to checkmate in order to solve the puzzle. Puzzles vary depending on checkmate that needs to be done in certain number of moves. The difficulty will also vary depending on how many pieces and what kind of pieces are in the puzzles. Since different pieces have different patterns and limitation of movement, the puzzle's difficulty is higher for the algorithm if there are pieces with high mobility like the queen versus low mobility such as pawn. We measure the performance of the search algorithm by its speed and quality. Quality will be defined as the strategies picked by the algorithm that solves the puzzle or gives it the higher probability to solve the chess puzzle.

This problem is interesting because chess has lots of states and we are trying to find solutions through streamlined algorithms and techniques. The environment of chess game is known, fully observable, multi-agent, stochastic, static and sequential. Fully explored chess game is very expensive and it takes forever as it has 10$^{(}$47) states and 10$^{(}$123) trees. It would not be feasible to consider all the options. So we will look at simple chess puzzles with less pieces. The problem is to find different search algorithms and techniques that will give us efficient way with high performance based on its speed, space allocation and quality of the strategies it takes given our limitations. Puzzles will vary in difficulty based on checkmate in 1 moves, checkmate in 2 moves, checkmate in 3 moves and total number of pieces in the board. We will get the puzzles and their solutions from www.chesspuzzles.com.

2 Related Work

[1] The complexities of various search algorithms are considered in terms of time, space, and cost of solution path. It is known that breadth-first search requires too much space and depth-first search can use too much time and doesn't always find a cheapest path. A depth-first iterative-deepening algorithm is shown to be asymptotically optimal along all three dimensions for exponential tree searches. The algorithm has been used successfully in chess programs, has been effectively combined with bi-directional search, and has been applied to best-first

heuristic search as well. This heuristic depth-first iterative-deepening algorithm is the only known algorithm that is capable of finding optimal solutions to randomly generated instances of the Fifteen Puzzle within practical resource limits.

[2] Reinforcement learning is the learning of a mapping from situations to actions so as to maximizes a scalar reward or reinforcement signal. The learner does not need to be directly told which actions to take, as in most forms of machine learning, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, an action may affect not only the immediate reward, but also the next situation, and consequently all subsequent rewards. These two characteristics—trial and error search and delayed reward—are the two most important distinguishing characteristics of reinforcement learning.

[3] Classification of different strategies based on attributes would help to improve the performance of search algorithm during the chess game. Methods for voting classification algorithms, such as Bagging and AdaBoost, have been shown to be very successful in improving the accuracy of certain classifiers for artificial and real-world datasets. The paper reviews these algorithms and describe a large empirical study comparing several variants in conjunction with a decision tree inducer (three variants) and a Naive-Bayes inducer. The paper helps us understanding of why and when these algorithms, which use perturbation, reweighting, and combination techniques, affect classification error.

[4] Bjornsson's paper provides experimental results with the new pruning method in the domain of chess. The eciency of the -algorithm as a minimax search procedure can be attributed to its eective pruning at so-called cut-nodes;

ideally only one move is examined there to establish the minimax value. This paper explores the benets of investing additional search eort at cut-nodes by also expanding some of the remaining moves. The paper says that the results show a strong correlation between the number of promising move alternatives at cut-nodes and a new principal variation emerging. Furthermore,a new forward-pruning method is introduced that uses this additional information to ignore potentially futile subtrees.

[5] In his seminal work on chess programs, [Shannon] introduced a lower bound on the game-tree complexity of chess as well as an evaluation function of the form:

$f(P) = 200(K - K') + 9(Q - Q') + 5(R - R') + 3(B - B' + N - N') + (P - P') - 0.5(D - D' + S - S' + I - I) + 0.1(M - M')$

— K,Q,R,B,N,P are the number of white kings, queens, rooks, bishops, knights and pawns.
—D,S,I are doubled, backward and isolated white pawns.
—M is white mobility (measured, say, as the number of legal moves available to white).
—The prime values are the corresponding values for black.

[6] [Atkin and Witten] presents a model for evaluating chess positions that is based on the mathematical relationship between the chess pieces and the squares on the chessboard. In this model, a piece and the squares that it "attacks" are labeled a "simplex," and simplexes are composed into "complexes." By analysis of the simplexes and complexes, Atkin and Witten showed that it is possible

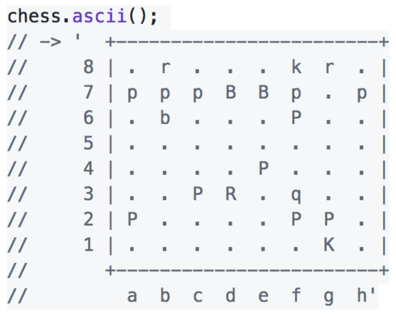
to make reasonably effective move predictions using real games as a point of comparison. Their work shares some commonalities with the approach taken in this paper, but their primary focus is on the geometry of the relationships as opposed to the network qualities of those relationships.

[7] According to [Elkies], Normally a chess problem must have a unique solution, and is deemed unsound even if there are alternatives that differ only in the order in which the same moves are played. In an enumerative chess problem, the set of moves in the solution is (usually)unique but the order is not, and the task is to count the feasible permutations via an isomorphic problem in enumerative combinatorics.

[8] [Fernandez and Salmeron] experimented with search tree heuristics based on accepted piece and position values modified by a shallow Bayesian network built to adapt the engine's play to that of its partner. Using the Bayesian network, the engine attempts to classify the opponent's playing style and select a complementary playing style given the current stage of the game.

3 Approach to Solve Problem and Experiment Design

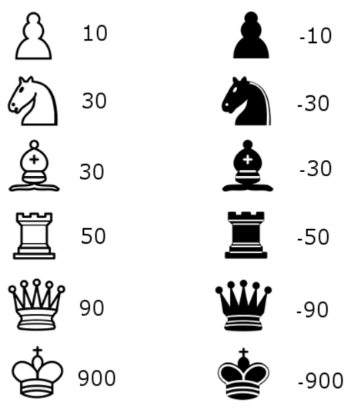
To tackle the problem, we need to first setup how to visualize the chess board and how to move the pieces in it. We used Github's open API chees.js by jhlwa. It had all the ingredients to setup the board in either a simulated visualized real chess board or in a matrix form of 8x8 with numbers and letters. In order to simulate the real chess board, the matrix form was the input. To make things easier, we only used the matrix form that looked as follows:



```
chess.ascii();
// -> '  +------------------------+
//     8 | .  r  .  .  .  k  r  . |
//     7 | p  p  p  B  B  p  .  p |
//     6 | .  b  .  .  .  P  .  . |
//     5 | .  .  .  .  .  .  .  . |
//     4 | .  .  .  .  P  .  .  . |
//     3 | .  .  P  R  .  q  .  . |
//     2 | P  .  .  .  .  P  P  . |
//     1 | .  .  .  .  .  .  K  . |
//       +------------------------+
//         a  b  c  d  e  f  g  h'
```

The capital letters are for the white pieces and small letters are for the black pieces. P represents Pawn, B represents Bishop, K represents Knight, R represents Rook, Q represents Queen and K represents King. The move function was already built in the chess.js framework, that would allow me to test different move functions on various chess pieces to solve chess puzzles. It also had additional options of non-capture, a pawn push of two squares, an en passant capture, a standard capture, a promotion, kingside castling and queenside castling. In order to reduce complexity, We will be picking puzzles that doesn't require en passant capture, promotion, kingside castling and queenside castling to solve them. The move function also checks for what moves are valid, so pieces don't need to try every spot on the board but only legal moves that they are allowed to do.

Next thing we need to do is to evaluate the power of different pieces, we do so by assigning general standard values that's usually used in chess assessments.

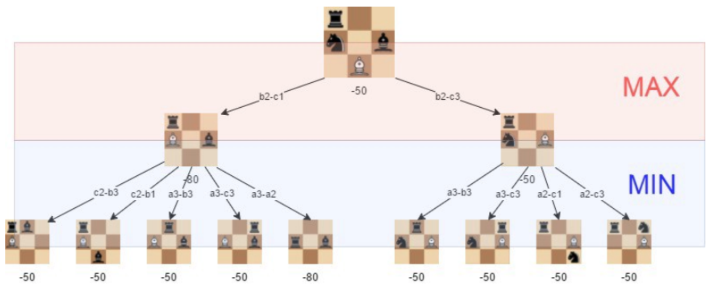


We will be solving the problem of chess puzzle with simple minmax search and Alpha-beta pruned minmax search. Minmax is a decision-making algorithm that is commonly used in two player games that are turn-based. The algorithm's goal is to find the best next move or the optimal next move. The algorithm describes one player as the minimizer and another player as the maximizer. As given in figure above, we assign value to the board. Here, white pieces are given positive values and black pieces are given negative values. In our case, the player with black pieces tries to minimize the total score whereas the player with white pieces tries to maximize theirs. Our game is based on the concept of zero-sum game. In other words, one player gain is another loss and one player loss is another player's gain. For minmax algorithm, we start at the root node and choose the maximum possible node. We recursively reach leaves with scores and back propagate the scores. The min nodes would choose score with the lowest values.

In a normal chess game, the initial total score would be zero for the board as both players would have equal amount of pieces in the board. However, that's not the case for chess puzzle. We will have different numbers and types of chess pieces assigned to each players. For our problem, we will only assume that the algorithm is assigned white piece and it's trying to maximize its score.

Alpha-beta pruning for minmax algorithm, helps us shorten the branches in the search tree by cutting some of the branches off and disregarding the subtree rooted from that branch. It cuts off branches in the game tree which need not be searched because there already exists a better move available. It is called Alpha-Beta pruning because it passes 2 extra parameters in the minimax function, namely alpha and beta. Beta is the minimum upper bound of possible solutions. Alpha is the maximum lower bound of possible solutions.

To solve the puzzle, our algorithm has to get 900 points added in it's favor. For check mate in 1 puzzle it has to do so in 1 move. The minmax search tree would be of two level max-min. For checkmate in 2, it has to make a move then wait for another player to make a move then gain 900 points in it's favor by the next move. The minmax search tree would be of four levels max-min-max-min. Similar case of checkmate in 3, where it has to gain 900 points by it's third move. The minmax search tree would be of six levels max-min-max-min-max-min. The puzzles are setup in such a way that checkmate 2 can't have checkmate in 1. And checkmate in 3, would need at least 3 moves to do checkmate.



Here is a figure of limited minmax search tree with two level max-min. In our cases, the second level would have all the possible moves that the black pieces would be able to make and third would have their respective possibilities of moves that white pieces can make and so on. There would be lot more than two branches depending on the complexity of the puzzle.

Writing algorithm such as breadth-first search or depth first search only, with condition that iterates until it satisfies +900 condition wouldn't be practical in real chess games as chess has 10$^{(}$47) states and 10$^{(}$123) trees. Even though states and trees are less than a normal chess game for the puzzles, it's still lot more to deal with than breadth-first search and depth first search. It wouldn't be efficient to go through all these states and trees to satisfy that condition. In addition, since two player play the game, algorithms next moves is highly dependent on opponents move , making things more complicated.

Our algorithms will evaluate the board value by adding all values of pieces that it has and comparing it with it's opponent. Its goal would be to reach its own initial board value plus 900. The time for the algorithm will be assessed by putting a start timer function in the beginning of the algorithm and end timer once it solves it. Regarding allocation we will be checking the size of different matrixes of board that it allocates in the buffer to solve the problem. We have three puzzles each (1a, 1b, 1c), (2a, 2b, 2c) and (3a, 3b, 3c) for checkmate 1, 2 and 3 respectively.

4 RESULTS

Checkmate in 1 (time in seconds, memory usage in bytes)

	Chess puzzle 1a	Chess puzzle 1b	Chess puzzle 1c
Simple Minmax	(2.232, 226)	(4.841, 352)	(2.193, 192)
Alpha-Beta pruned Minmax	(2.234, 226)	(4.847, 352)	(2.199, 192)

Checkmate in 2 (time in seconds, memory usage in bytes)

	Chess puzzle 2a	Chess puzzle 2b	Chess puzzle 2c
Simple Minmax	(9.382, 3751)	(6.043, 2941)	(12.252, 472)
Alpha-Beta pruned Minmax	(6.150, 2333)	(4.969, 1972)	(8.282, 3.080)

Checkmate in 3 (time in seconds, memory usage in bytes)

	Chess puzzle 3a	Chess puzzle 3b	Chess puzzle 3c
Simple Minmax	(22.928, 6102)	(5.956, 1092)	(73.686, 10829)
Alpha-Beta pruned Minmax	(16.976, 3649)	(3.723, 961)	(39.352, 7028)

5 Analysis

In the case of checkmate in 1, it was only one turn, so there can be no pruning done. Hence for both cases they have similar time to solve the problem and same bytes used for the space. There was slightly better time by simple minmax compared to alpha beta pruning because there were no pruning done, however alpha beta pruning has two more parameters to consider in each calculation which might have resulted in this uptick. We defined quality as the ability to provide the solution to the problem. The quality of the result is the same since both of them solved the problem.

In the case of checkmate in 2, alpha beta pruned minmax was clearly better in terms of time used and space allocated in all three chess puzzles 2a, 2b and 2c. All these puzzles had different amount of pieces and types in it. Both algorithms provided solution to the problem so they have same quality.

In the case of checkmate in 3, alpha beta pruned minmax was again clearly better in all three puzzles 3a, 3b and 3c in terms of time and space used. These puzzles also varied in amount of pieces and types in it. Both algorithms are of same quality as both provided with solution.

From the data tables of checkmate in 1, 2 and 3, we clearly observed that alpha-beta pruned minmax is superior to simple minmax in terms of speed and space with both providing the same quality result. We also observed that degree of improvements is dependent on the structure of the problem. With alpha-beta, we get a significant boost to the minimax algorithm. The alpha-beta pruning does not influence the outcome of the minimax algorithm, it only makes it faster. The alpha-beta algorithm also is more efficient if we happen to visit first those paths that lead to good moves. It is indifferent if there is no pruning happens. Pruning is likely to happen with many turns as there are lots of branches and nodes for comparison.

6 Conclusion

We solved the chess puzzle by simple minmax and Alpha Beta pruning of minmax. We concluded that Alpha Beta minmax is better in terms of speed and space allocation and gives same results. We tried both algorithms with variation of puzzle types with checkmate in 1, 2 and 3 with different numbers and types of pieces. All concluded the same result. There was one exception where there was no pruning needed to be done in solving the problem of checkmate 1. In that case, the result was indifferent. Breadth-first search or depth first search only, with condition that iterates until it satisfies solution condition would not be practical in real chess games as chess has $10^(47)$ states and $10^(123)$ trees. Minmax is better approach than Breadth-first search and depth first search.

However, when it comes to exceptionally high branching factor like GO game, Minmax might not be the best choice of algorithm. Nonetheless, given a proper implementation, it can be a streamlined approach to solving chess problems or playing chess. For real chess game it won't be feasible to construct the whole game tree. So building partial tree based on different strategies learned from Artificial Neural Networks giving weight to those strategies would be a better addition to the implementation of the alpha-beta pruning minmax algorithm.

References

[1] Richard E. Korf. *Depth-first Iterative-Deepening: An Optimal Admissible Tree Search.* 1985.

[2] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning I: Introduction.* 1998.

[3] Eric Bauer and Ron Kohavi. *An Empirical Comparison of Voting Classification Algorithms: Bagging, Boosting, and Variants.* 36(1):105–139, 1999.

[4] Yngvi Bjornsson and Tony Marsland. *Multi-Cut $\alpha\beta$-Pruning in Game-Tree Search.* Theoretical Computer Science, vol. 252(1-2), 177–196.

[5] Claude E. Shannon. *Programming a Computer for Playing Chess.* Philosophical Magazine, Ser.7, Vol. 41, No. 314 - March 1950.

[6] R. H. Atkin and I. H. Witten. *A Multi-dimensional Approach to Positional Chess Int.* J. Man-Machine Studies (1975) 7, 727-750.

[7] Noam D. Elkies. *New directions in enumerative chess problems.* 2005.

[8] Fernandez, A. and Salmeron, A. *BayesChess: A computer chess program-based on Bayesian networks.*. Pattern Recognition Letters archive, Volume 29 Issue 8, June, 2008 Pages 1154-1159.