AI Agent for Chinese Chess

Li Deng

2016 Autumn Stanford CS221

Abstract

This project aims to implement an AI game engine for Chinese chess, which is a popular board game in China. Different from Chess, Chinese chess has more complex rules and larger branching factor, making it more challenging to simulate the game, to evaluate states accurately and to search the game tree efficiently. To tackle the three problems, a simulator with web interface and server is carefully designed; several different strategies are implemented to play against each other with live game analysis.

First, *Greedy* strategy is used as base case. To increase efficiency of game tree search, *Minimax*, $\alpha - \beta$ *Pruning* with or without move reordering, and *MonteCarlo Tree Search* are implemented. To improve evaluation of game states with function approximation, *Temporal Difference Learning* is explored to tune feature weights as a reinforcement learning approach. Winning rates are compared among different strategies. Based on the analysis, the AI agent is evaluated against human player with satisfactory performance and efficiency.

Keywords: Chinese Chess, *Minimax*, $\alpha - \beta$ *Pruning*, *MCTS*, *TD Learning*

1. Introduction

Chinese chess is one of the most popular board games in China, which dates back to Song Dynasty (1127−1279 A.D.). It is a two-player, zero-sum game with complete information. The board represents the battle field between two armies, with one river across in the middle. Each team has one *King*, two *Horses*, two *Elephants*, two *Advisors*, two *Rooks*, two *Canons*, and five *Pawns*. The team who captures the opponent king first wins the game. Detailed game rules can be found at *Wikipedia*

The expert knowledge of Chinese chess started to be developed some 800 years ago. Competitions between computers against human are held annually in Asia, where the intelligence level of computers in player Chinese chess achieves the level of the best human player. However, there is no such open-source AI powered Chinese chess engine available to public. So this project seeks to explore implementing the

AI engine based on what is learned in CS221 at Stanford this Autumn.

In terms of complexity, The state-space complexity of Western chess and Chinese chess was estimated by Allis (1994). The game-tree complexity of Chinese chess is based on a branching factor of 38 and an average game length of 95 plies (Hsu, 1990).

Table 1 Complexity of Chinese Chess

Game	Space Complexity	Game Tree Complexity	Branching Factor
Chess	50	123	35
Chinese Chess	**48**	**150**	**38**
Go	160	400	250

2. Simulator Design

2.1. State-Based Game Design

The game implementation takes a fully Object Oriented approach, which contains the three classes: *State*, *Agent* and *Piece*. ***State*** captures all the information needed to describe the game status at a certain time point. Instance of State contains two ***Agent*** instance, representing the two teams in the state respectively. Agent is the parent class of several different strategies. Shared methods like moving a piece, updating legal moves, are implemented in the class of Agent, while each strategy has its own method to *compute next move*, where each strategy has its own policy to propose a move for the current state. Instance of Agent contains a list of ***Piece***s, each has the attribute of *Position* and *Name*. Each time a move is made by an agent, the state is automatically updated, and each agent calls its *update_legal_moves* to refresh the web interface. This State-based design with Object Oriented approach captures essential information about the game, making the implementation succinct and elegant. The core of the engine is then just to implement different *compute_next_move* methods for each agent based on its policy.

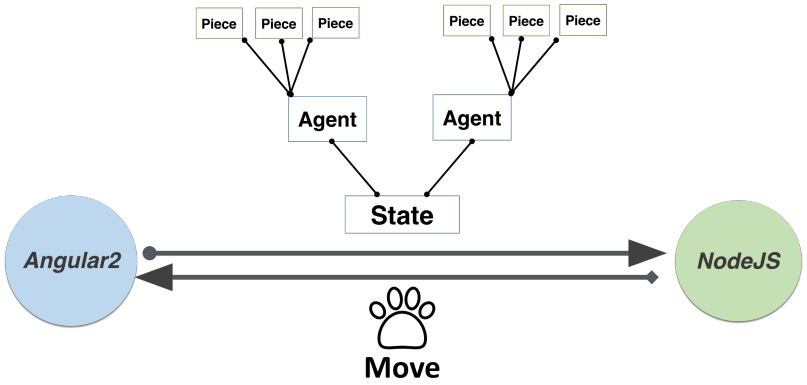


Figure 1 Simulator Architecture

2.2. Platform

Web is used as the game interface for its flexibility and user-friendliness. Two games modes can be selected by the user: *Human Mode* and *Simulation Mode*. In *Human Mode*, human player plays against the computer with selected strategy and search depth. Clicking on a piece, the legal moves for that piece are highlighted.

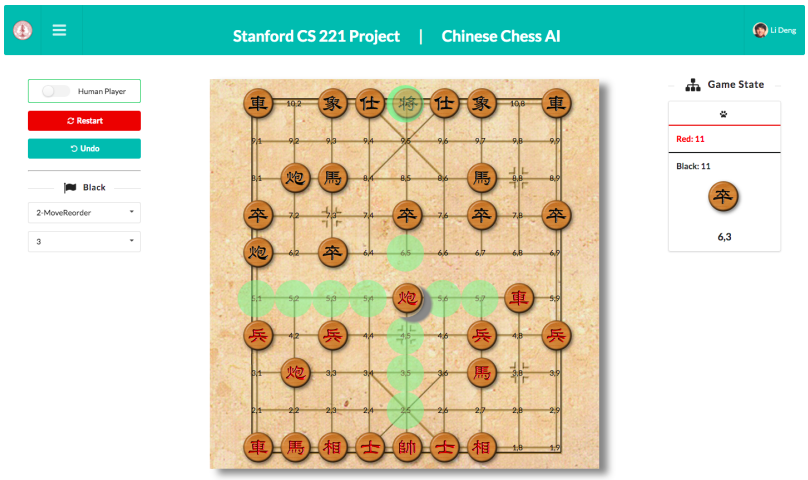
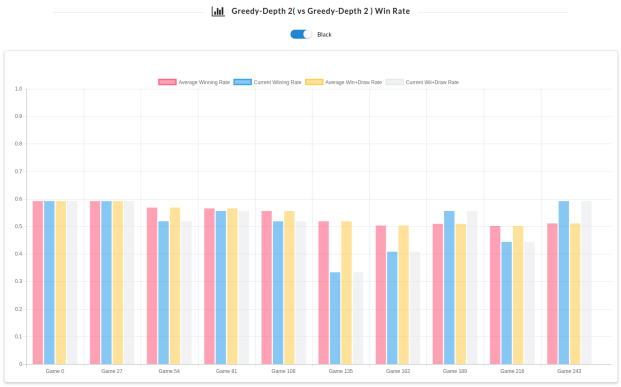


Figure 2 Overview of Simulator

In *Simulation Mode*, two policies play against each other for multiple games by the computer, with selected policy and search depth for each team. Live winning rate for each team for all the games played so far is plotted below for comparison and analysis of learning performance(if *TD Learning* is one of the playing policies). As base case, winning rate for *Greedy* VS *Greedy* (converging to 0.5 in 240 games) is shown:



In the front-end, *Angular2* and *SemanticUI* builds the framework. Light-weight computation like updating state to refresh legal moves are taken by the front-end browser. After a move is made, and next move is expected from the computer, the instance of current State is sent to back-end to take the overhead of computation. *Node.JS* is the back-end server, which shares the same compiler as the browser, so

the same code can be shared by both front-end and back-end. Instead of coding directly in *Javascript*, ***Typescript*** is the implementation language, for its much better Object Oriented support, and more elegant syntax, which will be trans-compiled to *Javascript* at runtime. The carefully designed architecture not only has the advantage of using one language across front-end and back-end, but also has remarkable efficiency, since both *Node.JS* and *Chrome* use the famous *Google V8* engine, which runs approximately 6.3 times faster than *python*. With the infrastructure designed and built, now the core of the AI engine, exercising different policies to compute next move, is then implemented.

3. AI Engine: Policies to Compute Next Move

3.1. Base Line: ***Greedy***

All pieces are assigned with a value indicating their importance. **Greedy** policy picks the move from all legal moves that captures the most important opponent piece if there exists at least one capture move, otherwise just takes a random move. This is the most simple and naive approach, used as the base line for all policies.

3.2. ***Minimax***

In some sense, *Greedy* only looks at the first level of the game tree. Now we extend the game tree search to a certain depth by using the standard **Minimax** search policy. The first step is to define a better evaluation function of the state. *Greedy* agent only considers the value of the piece itself. Apparently this is not enough, since even the same piece may have completely different value at different positions. One typical example is that once a pawn crosses the river into the opponent field, it becomes much more important due to increasing attacking power and flexibility by the game rule. Based on existing literature, the matrix of positions values are assigned to each kind of piece (pieces of the same kind shares the same position values). By function approximation, the evaluation of the state is defined to be the difference of summed values between the two agents:

$$Eval(state) = \begin{cases} +\infty & \text{Red Team wins} \\ -\infty & \text{Black Team wins} \\ V_{red} - V_{black} \end{cases}$$

where

$$V_{agent} = \sum_{piece_i \in agent} (pieceVal(piece_i.name) + posVal(piece_i.name, piece_i.position)$$

Based on the evaluation function, the search is then straightforward by using the standard *Minimax* search policy.

3.3. $\boldsymbol{\alpha - \beta}$ ***Pruning***

Consider an extreme case: if we can already capture the opponent king in this move, then we just take it, instead of bothering to consider any other moves. This is a typical case when we can bound our search with an upper bound β for the *Min* levels and an lower bound α for the *Max* level, to cut off unnecessary branches to search the game tree more efficiently. This is the intuition behind $\alpha - \beta$ Pruning. With the pruning aided, to search a depth-3 tree to compute a move, average run-time decreases from 3020 ms to 920 ms.

3.4. $\boldsymbol{\alpha - \beta}$ ***Pruning with Move Reordering***

Ordering of the moves to consider in $\alpha - \beta$ Pruning significantly affects the pruning effect. In worst case, $\alpha - \beta$ Pruning still has to search every node, taking $O(b^{2D})$, where D is the number of plys; while with the best ordering, $O(b^D)$ can be achieved. So this policy reorders the moves before diving into searching the branches at each level.

Consider the sequence of consideration when a human player plays Chinese chess. He or she will first consider the moves that can checkmate, then the moves that capture important opponent pieces, and then moves that capture less important opponent pieces, and lastly the empty moves. Based on the observation and intuition, the same ordering is used to reorder the moves in this policy. This policy is proved to be surprisingly fast: for searching a depth-3 tree to compute a move, average run-time decreases from 920 ms to 580 ms, with the same result compared to $\alpha - \beta$ Pruning without move reordering. More detailed comparison are listed in the Result section.

3.5. ***Monte Carlo Tree Search***

Except *Greedy* policy, all the following three policies aforementioned are all based on *Minimax* search framework. They are all deterministic policies, taking a move purely by **exploitation**, without **exploration**, which is limited in search depth and number of states evaluated.

On the contrary, *Monte Carlo Tree Search (MCTS)* takes a heuristic search approach by prioritizing the more promising moves and interpolates between *exploitation* and exploration. As a node in the search tree, each state has two more parameters: **visits** and **sum_score**, both of which are initialized to be 0. *sum_score* is the total score obtained so far for this state, which is the sum of scores for all of its child states visited. To encourage exploration, the evaluation of state is now based on:

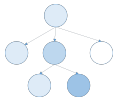
$$f = \frac{V}{n_i} + c \cdot \sqrt{\frac{ln(N)}{n_i}}$$

where:

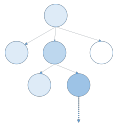
$$V : \textit{sum\_score} \text{ accumulated so far}$$
$$n_i : \text{number of } \textit{visits} \text{ for the node}$$
$$N : \text{number of } \textit{visits} \text{ for its parent}$$
$$c : \text{exploration parameter}$$

Instead of *search depth*, here *MCTS* is parametrized by *Number of Simulations*. And in each simulation, 4 steps are recursively called:

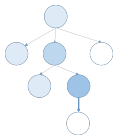
- **Select**: From root all the way down, it a child node that has not been visited before ($visits = 0$), then return the node; otherwise pick the node that has the highest evaluation value f, and recursively call *Select* on its child nodes, until we reach a node that has not been visited before, and return the node;



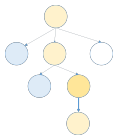
- **Simulation**:From the *Selected* node, take one greedy move, and evaluate the successor state;



- **Expansion**:Add the *simulation* node to the tree;



- **Back-propagation**:Update all the parent nodes of new added node by increasing *visits* and *sum_score*



The four steps are recursively called to collect as many as statistics about moves as possible, given a limit on total number of simulations. For the *exploration parameter* c, in standard *MCTS* implementation, it is usually set to be $\sqrt{2}$ times number

of wins. In our implementation, it is set to an equivalent $\sqrt{2}$ times of the piece value of the king.

It is observed that:

- More simulations increase the winning rate of *MCTS*, which is expected;
- Larger *exploration rate* adds more randomness to the policy, while smaller *exploration rate* lends more weights in exploiting existing evaluations. A suitable value of *exploration rate* is essential for the balance between exploitation and exploration, and leads to more chance in winning.

3.6. ***TD Learning***

All the policies above aim to search the game tree more efficiently. And now we turn to *TD Learning* to get a more accurate evaluation of the states by taking a reinforcement learning approach.

Considering that so far evaluation of the state only depends on single piece's value and position, not taking into account of the relations between pieces. So the following features are incorporated from two perspectives:

- **Attacking Power**: number of threatening moves, number of capture moves, number of center cannons, number of aligned cannons;
- **Mobility**: number of possible moves of rook, horse, elephant

A certain number of games are played between an agent with *TD Learning* policy and other agent with a different policy. For each game, features vectors for each state throughout the game are stored as a matrix W. After the end of each game, a reward is obtained to update the weight vector:

$$w^{t+1} = w^t + \eta \cdot r^t \cdot (A \cdot \overline{\phi^t} + B \cdot \phi^{t*} - w^t)$$

where:

w^t :weights vector at game t

$\overline{\phi^t}$:average feature vector for all states in game$t$

ϕ^{t*} :last(determining) feature vector in game$t$

η :learning rate $\qquad \eta = \dfrac{1}{\sqrt{t}}$

r^t :reward at game t. $\qquad r^t = \begin{cases} 1 \text{ win} \\ -1 \text{ Lose} \\ 0 \text{ Draw} \end{cases}$

The update equation is based on the intuition that both the states across the game and the last state(which tells a lot about whether the agent wins or loses) affects the evaluation of features.

4. Results

4.1. Search Efficiency

Search Effenciency Comparison

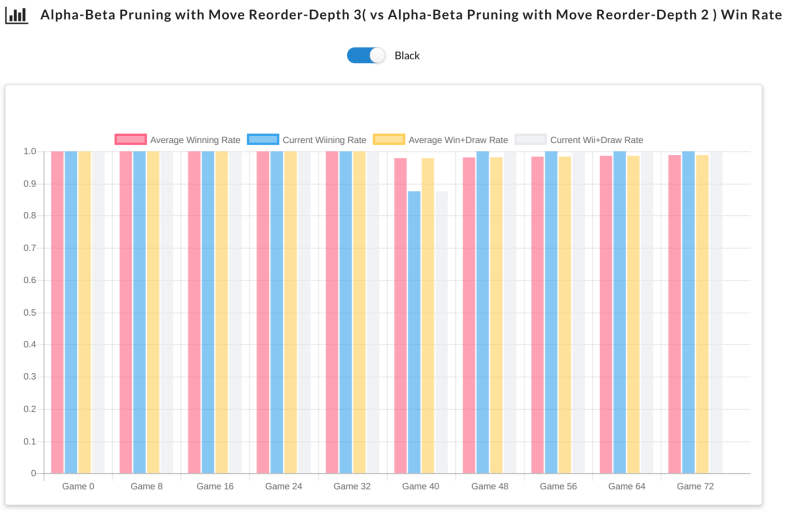
Strategy	Search Depth(Simulations if MCTS)	Average Runtime for Each Move(ms)
Greedy	1	4
Alpha-Beta Pruning with Move Reorder	2	64
Alpha-Beta Pruning	2	76
Alpha-Beta Pruning with Move Reorder	3	237
Alpha-Beta Pruning	3	565
Alpha-Beta Pruning with Move Reorder	4	1895
Alpha-Beta Pruning	4	6756
Monte Carlo Tree Search	1000	611
Monte Carlo Tree Search	2000	1173
Monte Carlo Tree Search	3000	1743
Monte Carlo Tree Search	4000	2798

From this table, we can see that:

- Not surprisingly, *Greedy* is the fastest policy;
- To search the same depth, reordering the moves greatly speeds up the search compared to $\alpha - \beta$ Pruning without reordering, especially when search depth becomes larger;
- *Monte Carlo Tree Search* is pretty fast for thousands of simulations

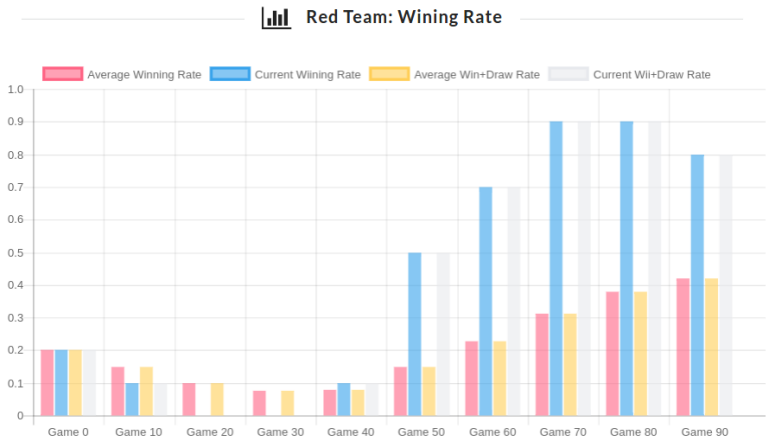
4.2. Intelligence Level

- *$\alpha - \beta$ Pruning with Move Reorder: Depth 3 VS Depth 2*



So it is shown that with deeper search depth, the agent becomes much more intelligent.

- *TD Learning Depth 3* VS $\alpha - \beta$ *Pruning with Move Reorder* Depth 3



We can see that the *TD Learning* policy becomes more intelligent over time after 100 games.

As a benchmark against the author, in 20 games where no *Undo* of a move is allowed (even though the functionality is provided in *Human Mode*), the *Move Reorder* policy has a winning rate of 0.55 with depth 4 and 0.8 with depth 5.

5. Conclusion

This project explores building the AI agent for Chinese chess, which is a challenging problem with complex rules and large branching factor. This project approaches the problem in two ways: to search game tree more efficiently by *Moved Reordering in* $\alpha - \beta$ *Pruning* and *Monte Carlo Tree Search*; and to evaluate state more accurately by *TD Learning*. A web-based simulator is designed for testing and analysis, and satisfactory results are obtained.

Further improvements may include:

- Using Neural Network to enhance the expression power of evaluation function: for this project, evaluation functions are all linear combination of features. Apparently this is not enough since it cannot capture the non-linearity between features. Also, features are just hand-crafted in this project. By using Neural Network to train the game, more features can be automatically selected, and non-linearity among features can be expressed;

- Open-game and end-game database: Chinese chess has thousands of years of history, and many people have summarized the best moves for open-games and end-games. If such a database can be built in the engine, it can greatly improve the performance in the opening and ending period in the game.

6. Reference

[1] Ong, Chin Soon, et al. "Discovering chinese chess strategies through coevolutionary approaches." 2007 IEEE Symposium on Computational Intelligence and Games. IEEE, 2007.

[2] Fang, Haw-ren, Tsan-sheng Hsu, and Shun-chin Hsu. "Construction of Chinese Chess endgame databases by retrograde analysis." International Conference on Computers and Games. Springer Berlin Heidelberg, 2000.

[3] Stanford CS 227B-General Game Playing: Lecture 8

7. Github Repo

The project is hosted on Github: https://github.com/dengl11/ChineseChessAI, with the project website https://dengl11.github.io/ChineseChessAI/.

The reader is welcome to git clone and have fun.