Chinese Chess AI Agent

Suting Chen, Zeen Chi[†], Bingnan Li[†], Zhongxiao Cong[†], Yifan Qin[†]
School of Information Science and Technology
ShanghaiTech University
No.393 Middle Huaxia Road
Shanghai, China 201210
{chenst,chize,libn,congzhx,qinyf1}@shanghaitech.edu.cn

Abstract

Introduction

Methods

In this section, we will introduce three methods we used in this project. They are Minimax Search, Reinforcement Learning, and Monte-Carlo Tree Search, respectively. We will briefly introduce their basic ideas and then explain how we implement them in our project.

Minimax Search

The most straightforward method for the adversarial search based game is minimax search algorithm. As taught in the class, the minimax search algorithm is a recursive algorithm that is used to find the optimal move for a player, assuming that the opponent also plays optimally.

Basic Idea The search tree is constructed based on the states and actions. Each state is a tree node, while each action a transit state s to s' is a tree edge. The root of the tree is a max layer, and for the following layers, the nodes are alternately min and max layers. The value of a node is the value of the state it represents.

However, it is impossible to build a full search tree for the chess board, because the complexity grows exponentially as we search deeper. Therefore, we utilize depth-limited search strategy, which means the searching process terminates when the depth of the tree reaches a certain value, for example, 3. Actually, as the game plays, there will be fewer pieces on the board and hence smaller branching factor, so we utilize iterative deepening based on the number of the pieces on the board. As the pieces become less, we search deeper.

Evaluation Function When searching process terminates, we have to evaluate the score of the state of the so-called leaf node. Based on (Yen et al. 2004), we have considered the following factors when designing the evaluation function:

- The power of each piece. We assign different values to different types of pieces, and the more powerful the piece is, the higher the value is. For example, we assign 600000 to General, 600 to Chariot, 450 to Cannon, 270 to Horse, etc.
- The position of each piece. We assign different values to different positions of the pieces, and the more advantageous the position is, the higher the value is. More specifically, we design a 10×9 matrix (which is the size of the board) for each piece, with each value representing the advantage of the position.
- The flexibility of each piece. Based on the moving range, different types of pieces have different flexibilities. We first assign different fixed values to each type, and calculate how many positions can each piece reach. We multiply the fixed value to the calculated value, and the result is the flexibility of the piece.
- The value of being threatened. We have to calculate how many piece can threaten the current piece. After the number is calculated, we can multiply it by the power of the piece.
- The value of being protected. It is also important to consider how many pieces can protect the current position, that is, if the current piece is killed by the enemy, how many allies can revenge for it. After the number is calculated, we can multiply it by the power of the piece.

For each leaf node state, we respectively calculate the weighted sum of the above factors of the player and the opponent, and the difference is the evaluation value of the state.

Optimization We apply alpha-beta pruning method to the minimax search algorithm, which can reduce the number of nodes that need to be evaluated. It stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

Reinforcement Learning

Since we need to know how to take each step, i.e., get the action, we use the Q-learning method.

Q value The Q value is $Q_k(s, a)$ where s denotes the current state of the game and a denotes the current action. We use the board to represent the current state of the game. Considering that the starting point and the landing point of this action will be different, we define the action as a tuple, which records which piece is picked and where this piece lands.

Reward function

Training design Considering the training time, we train our Qlearning agent as a red side with Random agent on the black side. After Qlearning agent makes a move, we record the current state of the board and its actions, and from these two we calculate the reward that the Qlearning agent gets in this move as r_1. Next, after the random agent makes its next move, we get the reward that the Random agent gets in this move as r_2. Then, at the end of this round, the actual reward that Red gets is $(r_1 - r_2)$. So the sample estimate is

$$sample = (r_1 - r_2) + \gamma \max_{a'} Q(s', a')$$

Then we update the q-value of $Q(s, a)$

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha[sample]$$

Since we need to explore as many different states as possible in the early stages of training, we introduced the ϵ-greedy function. Setting epsilon to 0.5 gives our agent a 50 percent chance of picking the optimal policy based on $\arg\max_a Q(s, a)$ and a 50 percent chance of taking the policy randomly.

Training Due to memory limitations, the training limit is 4000 epochs. The gamma is set to 0.8 and the learning rate is set to 0.8 for the first 1000 epochs, 0.5 for 1000 to 2000 epochs, 0.2 for 2000 to 3000 epochs, and 0.1 for 3000 to 4000 epochs. We write the explored qvalue to a .txt file in the form of .json. All values of the current $Q(s, a)$ are recorded once every 100 epochs.

Monte-Carlo Tree Search

It is known that the search strategy that AlphaGo uses is the Monte Carlo Tree Search(MCTS). The MCTS search algorithm can effectively solve the problem when the state space is too big. Although Chinese Chess does not have the search space as big as the Go game, where players can have hundreds of choices to put their chesses at each step, players still have about fourty choices to move their pieces, which indicates a large search space. So we think that it may be feasible to use the MCTS search algorithm in the Chinese Chess. As MCTS was not fully introduced in the lectures, here the basic idea of MCTS will be introduced and our implementation of MCTS will also be talked about.

Basic Idea MCTS search algorithm build the search tree node by node according to the outcome of the simulation results. To be more specific, the search tree is built during the iteration of four steps, Selection, Expansion, Simulation and Backpropagation.

In the Selection step, starting from the root node, the optimal leaf node will be choosed recursively until a leaf node L is reached. In the Expansion step, if the leaf node L that we reached is not the terminal node, we can have several child nodes of the leaf node. One of the child nodes of the leaf node L will be expanded and added to the search tree. Then in the Simulation step, we will start a simulated game from the child node C that we just expanded and get a final result. Finally in the Backpropagation step, we will back propagate the result from the child node to the root node of the search tree in order to update the movements or the choices sequence with the simulation result.

If only the estimated values of the simulation result of the nodes are utilized, chances are that the node choice of the Selection step will concentrate within a few nodes. Other nodes or the movements may be less explored. In order to avoid the concentration, both the estimated value and the visit time of the node is used to find the optimal leaf node.

Implementation

Results

In this section, we will demonstrate the results of three agents. We will first conduct multiple games between the three agents and the random agent, and then show the results of three agents playing against each other. Finally, we will select relatively the best agent to compete with the existing AI on the Internet to evaluate its effectiveness.

Our Agents v.s. Random Agent

Since a strategic agent is much better than a randomized algorithm, we show the superiority of the algorithm by making each agent play red and black, and then counting the win rate of our agent and the average number of moves needed to kill the opponent after 50 games.

Table 1: Win rate and average number of moves needed to kill the opponent

Red	Black	Winning Rate	Steps
Minimax	Random	100%	17.52
Random	Minimax	100%	12.84
RL	Random		
Random	RL		
MCTS	Random	100%	
Random	MCTS	100%	

Minimax v.s. MCTS

We also conduct 50 games between Minimax agent and MCTS agent to compare their superiority.

Table 2: Experiment results between Minimax and MCTS

Red	Black	Red Winning Rate	Draw Rate	Steps
Minimax	MCTS	78%	6%	36.94
MCTS	Minimax	42%	4%	39.57

Minimax v.s. Real AI

We also select the best agent to compete with the existing AI on the Internet. Since the existing AI is developed by a team of people for a longer time, and its strategy is much more high-level than ours, it is much more powerful than our agent. We only let our agent play red, and the existing AI play black. We conduct 20 games for them, and we play a role as an assistant to help moving the pieces. Hence we implemented a keyboard controller to enable mutually moving the pieces.

Table 3: Experiment results between Minimax and real AI

Red	Black	Red Winning Rate
Minimax	Real AI	15%

External Libraries

- `JUCE`, which is a cross-platform C++ framework for audio applications. It helps us to build the GUI of our project.
- `tkinter`, which is a Python library for GUI.

References

Yen, S.-J.; Chen, J.-C.; Yang, T.-N.; and Hsu, S.-C. 2004. Computer chinese chess. *ICGA journal* 27(1):3–18.