Tree Search Algorithms For Chinese Chess

Siyu Heng

Nanjing Institute of Technology, Nanjing, China

Abstract. Computer Games are an important aspect of Artificial Intelligence (AI) which has produced many representative algorithms of optimal strategies. Therefore, many AI researchers have applied game theory to various board games in an attempt to find the essence of games and grasp the core of AI behaviour. AlphaGo is a very famous and typical chess-playing program that uses Monte Carlo tree search (MCTS) to estimate the value of each node in the search tree and optimize the possible results. Chinese chess is one of the traditional chess versions with its typical strategy. However, due to its late start and high complexity, there are many more technical problems to be solved. In this paper, we recommend two commonly used search frameworks for Chinese chess: the minimax algorithm and the alpha-beta pruning algorithm. On this basis, to deal with the complex and changeable situation, the Chinese chess algorithm based on Monte Carlo tree search and appropriate neural network evaluation function is studied.

Keywords: Minimax, Alpha-beta pruning, Monte Carlo Tree Search, AlphaZero.

1. Introduction

Computer Games are considered to be one of the most challenging research fields in AI due to their highly complicated conditions and difficult analysis. Nowadays, with the rapid development of intelligent technology, computers have made great progress in completing battle-player games. For instance, IBM developed the “Deep Blue” series of chess-playing computers in 1991[1], and Alpha Go, a computer program, beats the top-level professional human players in 2016[2]. In computer games, the most classical problem is the path selection problem, which aims to find the optimal path of the game. The search tree (see Fig. 1) is a specific pattern for solving this problem. It is composed of limited space, movement choices, and defined rules. The nodes of a tree are the states, and each branch from root to leaves is considered a possible play. The transformation logic is described by rules, that is, the name of the game tree. However, traditional breadth-first or depth-first search methods prove to be exhaustive because they are non-heuristic path searches that require searching the entire tree in a fixed order until the target point is found. Therefore, the best choice for any selection in a search tree is uncertain. [3]

So, to improve the efficiency of information acquirement capacity, scientists have put forward many effective solutions to reduce search time in terms of width and depth. The first key strategy was proposed by Claude Shannon[4], the father of information theory, in 1950 by selectively traversing the branches of the tree to significantly reduce the number of searches. Nowadays, many new model searching algorithms have emerged to solve some concrete problems. By combining and extending different algorithms, great success has been achieved in many board games.

This paper mainly introduces the search algorithm of the Chinese chess game with minimax, alpha-beta pruning, and Monte Carlo Tree Search, respectively, and then discusses the neural network as the evaluation function to be incorporated into the tree search.

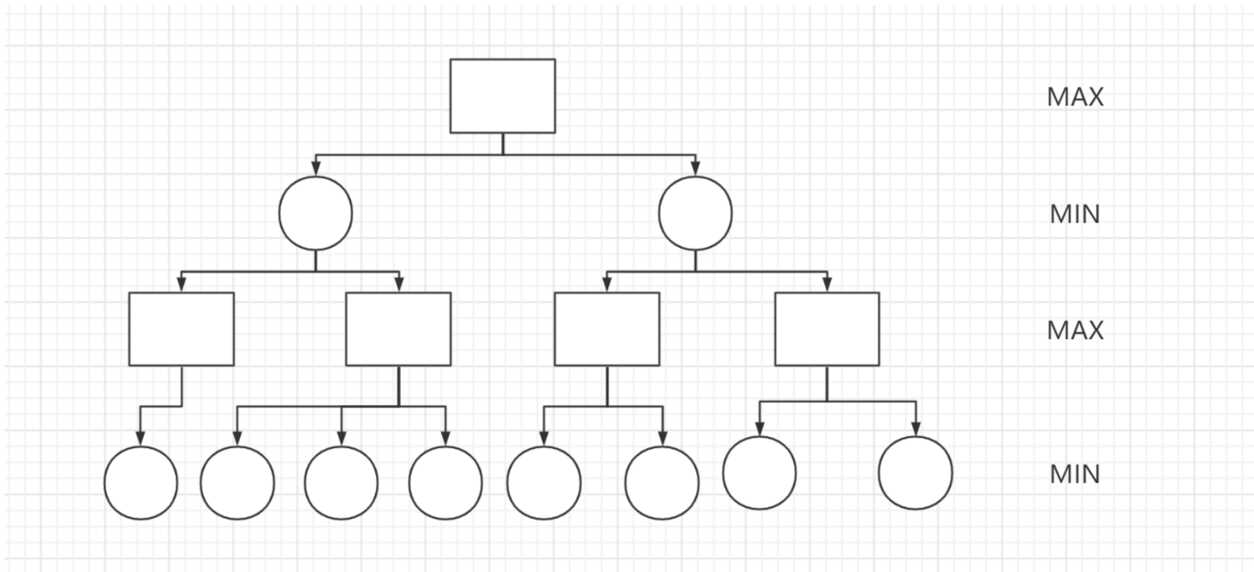

Fig. 1 Search tree

2. Background

Chinese chess is popular among young people because of its rich content, massive strategies, and flexible movements. It is a two-player strategy game played on a 9x10 game board, as shown in Fig. 2. The region is divided into two parts (i.e. black and red) by the river in the middle of the game board. Each side has sixteen pieces and seven kinds of pieces: Rooks, Knights, Cannons, Elephants, Warrior, King, and Pawns.

The rules are as follows: rooks, horses and cannons can be moved anywhere on the board. Elephants can only move within the player's own territory. Warrior and King are confined to a special region of *Jiugong*. There are two types of movement for pawns. After crossing the river, pawns can reach anywhere in enemy territory, but they can only appear on the top of a row or move in parallel when it is in their own side.

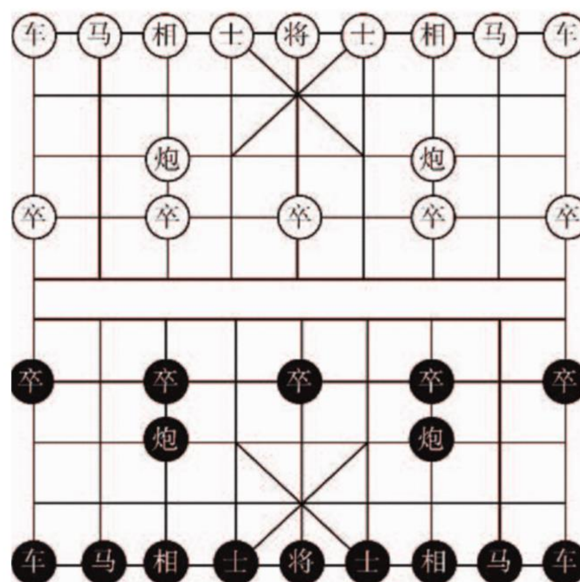

Fig. 2 Chinese chess board [5]

3. The search algorithm

A basic computer game system generally consists of evaluation functions, data representation, search algorithms and move generation. As for the Chinese chess game, each program searches before playing, trying to find out what is most advantageous in the current situation. Each branch represents a possible movement and the root of the search tree is the current position. The leaf nodes are evaluated by some static evaluation functions.

As the game processes, the level of the tree increases and more nodes need to be searched. Therefore, the size of the state space and the chess game tree can be very large, so it is impossible to access the entire tree in a limited time. There are different methods to overcome such a huge space and reduce exhaustive searching time: one is to limit the search depth while traversing the tree, such as minimax[6] search and alpha-beta pruning[7], and the other is to use the available limited sources to obtain an optimal result. Monte Carlo tree search[8] is applied for this purpose.

3.1. Minimax algorithm

The game tree is determined by both armies, and each side chooses what is best for itself. The layer that takes the evaluation function to get the maximum score is called the max layer. The opponent who chooses the situation of assessment of the lowest score to defend the counterpart from getting higher ranks is called the min layer. The max layers and min layers represent the decisions of different sides of the game.

The game tree in Fig. 3 is a minimax tree in Chinese chess, where the nodes represent the current position of the game and the edges represent the possible moves of the next position. The Square nodes indicate the position of the red player. Each circle node shows the node of the black player. The evaluation score is assigned to the pieces of the game board to analyze the current situation and indicate the advantage or disadvantage of red and black, respectively.

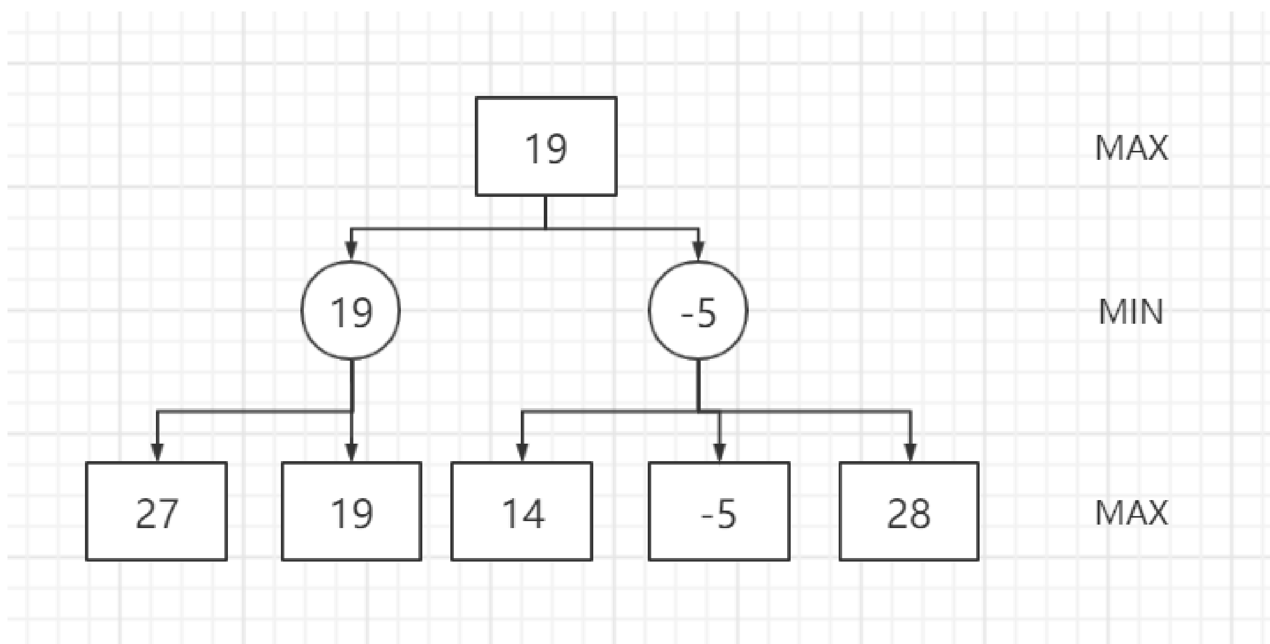

Fig. 3 Minimax Tree

The pseudocode in Algorithm 1 shows the main idea of using a Minimax algorithm in Chinese chess: each child is divided as the root of the subtree. If a leaf node is reached, it will be assessed by an evaluation function and its value will be returned to the parent node. The parent node will collect the results from all the child branches and then backtrack the optimal one to a higher node until finally returning to the root of the tree (i.e the current position). Later, the best position is found according to the evaluated value.

Algorithm 1: Minimax algorithm

```
1.  int miniMax(position p, int depth){
2.  //value means current value;
3.  // bestvalue means current optimal value;
4.  int bestvalue, value;
5.  // check if game is over
6.  if(GameOver)
7.      return evaluation(p); // return evaluation value;
8.  // if reached leaf node
9.  if(depth<=0)
10.     return evaluation(p);//return evaluation value;
11. // red player, initial optimal value is min;
12. if(p.color == red){
13.      bestvalue = -INFINITY;
14. }
15. // black player, initial optimal value is max;
16. else if(p.color == black){
17.      bestvalue = INFINITY;
18. }
19. // compare each movement value
20. for(each possible move m){
21.      MakeMove(m);  //execute movement
22.      value = miniMax(p,d-1);
23.      UnMakeMove(m);  // return to parent node;
24.      if(p.color == red){
25.          bestvalue = Math.max(value,bestValue);  // red player get the max value;
26.      else if(p.color == black)
27.           bestvalue = Math.min(value,bestvalue); // black player gets the min value
28.      }
29. return bestvalue; // return min/max value;
30. }
```

3.2. Alpha-beta pruning algorithm

Based on the traditional minimax algorithm, the alpha-beta pruning algorithm is proposed to improve the search performance. Instead of searching every possible solution in the entire state space, the alpha-beta pruning algorithm seeks to cut down the unnecessary branches evaluated by the minimax algorithm, and if the best move occurs first, the rest of the nodes can be ignored directly.

Optimization is performed during the traversal to see the effect of the alpha-beta pruning. The algorithm only explores the nodes with a high probability to achieve the goal but ignores the brunches that will never affect the final decision, thus improving the search speed. By not spending exploring the node whose value does not affect the final result, the algorithm avoids impractical time complexity and helps reduce execution time.

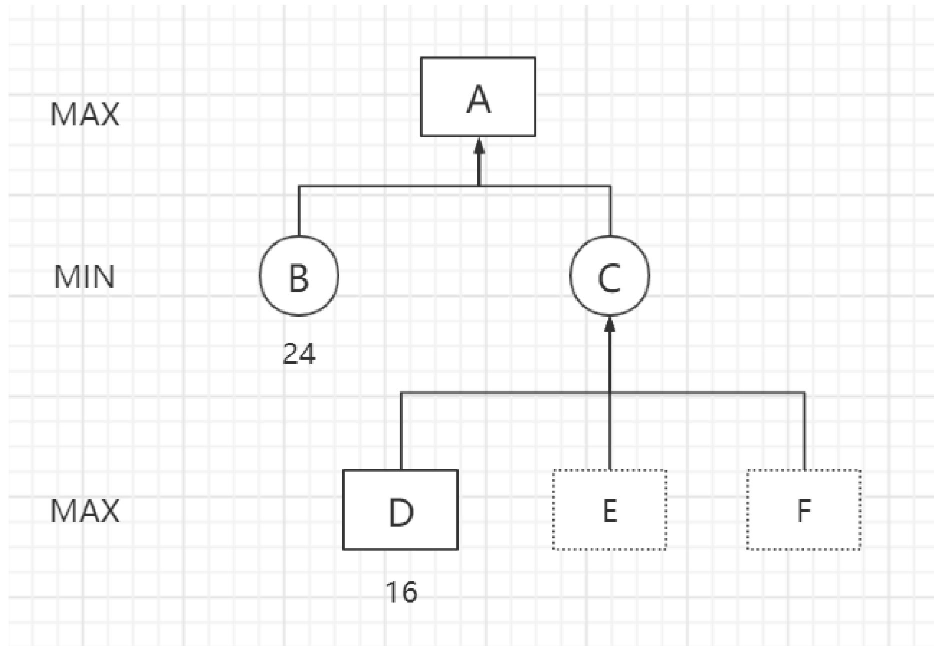

Fig. 4 Alpha pruning

The game tree in Fig. 4 represents a tree using alpha pruning. For the second layer (i.e min layer), B has a value of 24. Now, considering a cluster of nodes with root C, D has a value of 16. For C, in order to get a low score, its value should not exceed 16. Root A needs to choose the max value. Therefore, 24 is assigned to node A. It is noticeable that nodes E and F will not be evaluated because the optimal selection has been already produced and the previous value has determined which node would be chosen. No more moves need not be considered further.

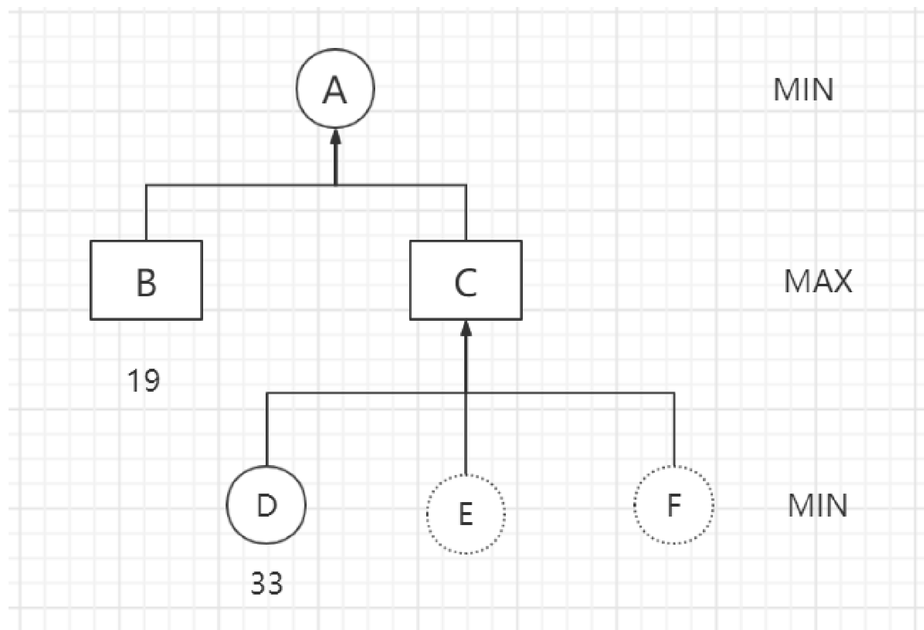

Fig. 5 Beta pruning

The structure of the beta pruning tree is shown in Fig. 5. For the max layer, the value of B and D is 16 and 33, respectively. To get the highest possible score, node C will be assigned a value of at least 33. Therefore, the value of node B will be assigned to A (i.e.19). During the whole beta pruning process, nodes E and F have been cut down(prune) to reduce the exploration space, as their value will not influence the final selection.

The implementation of alpha-beta pruning is shown in the following pseudocode in Algorithm 2.

Algorithm 2: Alpha-beta pruning

```
1.  int AlphaBeta(depth, alpha, beta){
2.       // check if game is over
3.      if(GameOver)
4.           return evaluation(p); // return evaluation value;
5.      // if reached leaf node
6.      if(depth<=0)
7.           return evaluation(p);//return evaluation value;
8.      if( is Min node){ //
9.           // compare each movement value
10.        for(each possible move m){
11.             MakeMove(m);  //execute movement
12.             score = AlphaBeta(depth -1, alpha, beta);
13.             UnMakeMove(m);  // return to parent node;
14.             if(score < beta){
15.                beta = score;
16.                if(alpha >= beta)
17.                     return alpha;
18.             }
19.         }
20.         return beta;
21.     }
22.     else{
23.         for(each possible move m){
24.             MakeMove(m);  //execute movement
25.             score = AlphaBeta(depth -1, alpha, beta);
26.             UnMakeMove(m);  // return to parent node;
27.             if(score >= alpha){
28.                 alpha = score;
29.                 if(alpha >= beta)
30.                      return beta;
31.             }
32.         }
33.       return alpha;
34.     }
35.  }
```

3.3. Monte Carlo tree search

Minimax algorithm and alpha-beta pruning are common methods for two-player game tree search. However, when it comes to some complex problems, a fixed-depth of tree search is not very efficient, because it can only predict a limited number of advanced situations and cannot see the effect of the state in the whole game. In consequence, it may overestimate or underestimate a value and miss the best reward that is out of the searching depth. Even if we can conduct a full search, it needs to spend enormous space and time. In this case, a Monte Carlo tree search(MCTS) is created to approximate the optimal solution and offset between the horizontal effect and vertical effect of a search tree.

Monte Carlo Tree Search has proved to be a very useful method for dealing with uncertain information. It is a search algorithm combining Monte-Carlo simulation and game tree search. MCTS has great advantages over traditional search methods in that it breaks down the complexity of the game space and treats the current state as a set of sampled results for that set only.

Monte Carlo Tree Search is a best-first search technique using simulation. Instead of traversing each node in the tree and evaluating their value, this method randomly selects samples from each part

of the search tree and develops some nodes iteratively according to the simulation results to grow the search tree. When a specified number of times or stop times are reached, the optimal decision is made to carry out the actual action in the game.

Fig. 6 shows four phases of MCTS for each search iteration: 1) *Selection*: In this step, MCTS selects the node with the highest value according to some tree policy. The game tree repeats traversing from the root node until the terminal node is met, and then the environment is moved to a new state. 2)*Expansion*: The second step is called expansion. Some new nodes will be added to the tree according to the current state. 3)*Simulation*: simulation is the next step. In this step, MCTS performs random actions for each player according to the fixed policy until the result is reached and a reword is obtained. 3)*Backpropagation*: When it comes to the backpropagation phases, the simulation result is propagated back to the node visited in the selection phase and updated to their statistic.

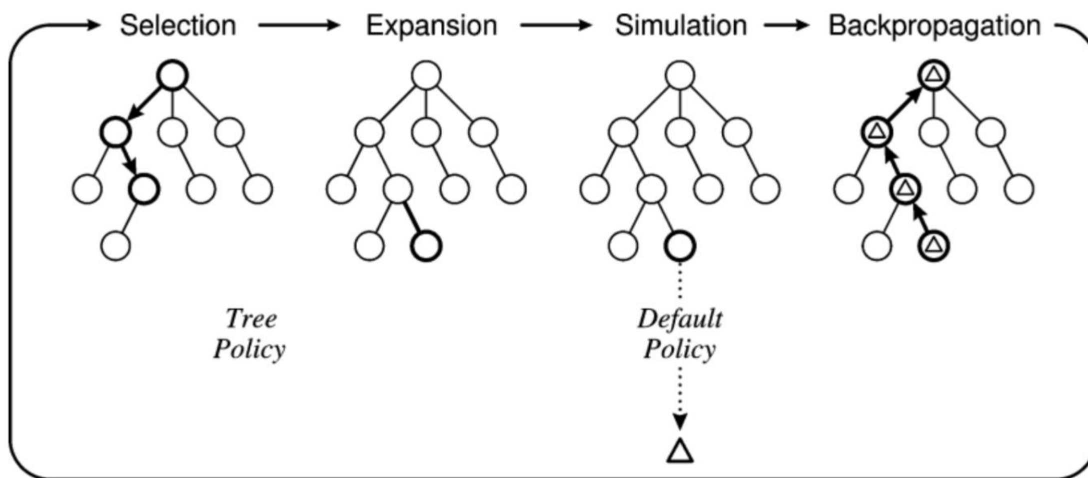

Fig. 6 One iteration of the MCTS approach.[8]

3.3.1 Selection

Upper Confidence bound(UCT)[9] is the most commonly used strategy applied in the selection phase while traversing the tree in MCTS. It is based on a multi-armed bandit problem[10]. For each depth, UCT selects the most promising node in a best-first manner. The effectiveness of a node is determined by two parts: the probability of winning for known actions and the node's potential to explore unseen or unknown actions. UCT creates new state-action values by using the following equation:

$$UCB_i = v_i + c \times \sqrt{\frac{2\ln(\sum_i T_i)}{T_i}}$$

Where v_i is the average reward of node $i;$ T_i is the number of times i has been selected; and c is a constant. We can use pseudocode in Algorithm 3 to estimate UCT value of each node and select best option for the state.

Algorithm 3: Selection phase of MCTS

```
1.  public TreeNode select_new(){   //using UCT to find the max value
2.         TreeNode selectedNode = null;
3.         double bestValue = 0;
4.         for (TreeNode c : children){      // calculate UCT value of each node
5.            double uctValue =c.totalValue/ c.nVisits //vi
                 +  Math.sqrt(2 * Math.log(nVisits)/nVisits)*explorationValue;
6.            if(uctValue>bestValue){
7.               selected=c;
8.               bestValue=uctValue;
9.            }
10.        }
11.        return selected;
12.    }
```

3.3.2 Tree Expansion

If arrival node t_0 describes a terminal state or is not fully expanded, a new leaf node t_1 will be added to the tree. An unaccessed action a corresponding to t_1 is then selected and applied to the current state *s*. The implementation is shown in pseudocode in Algorithm 4.

Algorithm 4: Expansion phase of MCTS

```
1.  public void expand(){ // expand tree node
2.        children=new TreeNode[nActions];
3.        for(int i=0;i<nActions;i++){
4.           children[i]=new TreeNode(); //initial the new node
5.        }
6.  }
```

3.3.3 Simulation

When starting from a new node, MCTS uses the default policy to play out the game until an outcome is produced. By using random simulation, the values of moves will become more accurate as numerous playouts will be explored to estimate the win probabilities of the final scores from that position so that these values will guide the policy toward the most promising moves.

3.3.4 Back Propagation

The main idea of backpropagation is to redistribute reward values or discrete results received at the end of a node to higher nodes in its iterative order. The number of visits to each node will increase, and the update of UCB value of each node will traverse the path, as shown in pseudocode in Algorithm 5.

Algorithm 5: Backpropagation phase of MCTS

```
1.  public void back_propagation(){
2.        for (!node.isRoot()){
3.           node.score += result;  //add node value
4.           node.visitCount++;  //add visited count
5.           node = node.parent;   // backtrack to parent node
6.        }
7.        node.visitCount++;
8.  }
```

3.4. AlphaZero Algorithm

The AlphaZero algorithm[11] combines Monte Carlo Tree Search with deep neural network(DNN) to learn without human knowledge and human heuristics[12]. Its considerable success demonstrates that reinforcement learning by self-play can reach superhuman performance and surpass the strength

of human champions in 2016. The performance of the AlphaZero draws researchers' great attention to other board games. Then a special program that can outperform any previously conventional approaches is created.

AlphaZero uses MCTS to guide self-play, thus improving its neural network. The search result is used as training data to make DNN clearer. Improved DNN helps MCTS play a better self-play and estimate the promising regional distribution of selection moves. Therefore, each component is used as an efficient input to adjust and enhance each other.

MCTS in AlphaZero contains three steps for each simulation, including *selection*, *expansion*, and *backpropagation*.

3.4.1 Selection

In the selection step, AlpahaZero is more like prediction + Upper Confidence Tree (PUCT)[13]. This strategy stores and updates exploration probabilities based on the neural network policies and the position and values. Like MCTS, AlphaZero can guarantee a perfect boundary between nodes that have not been fully accessed yet and the areas that appear to be promising. It is important to explore random gameplay to ensure that each node has the possibility of being selected. After the calculated budget is reached, AlphaZero selects a child node of the root with the highest visit count, which is considered to have the best result.

In this algorithm, the action of each node is always chosen by the formula:

$$a_c = \underset{a}{\operatorname{argmax}}\left(Q(s,a) + c_{puct}P(s,a)\frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}\right)$$

Where *s* is the state; *a* is an available action; *Q(s,a)* is the total value of action a from state *s; N(s,a)* is the number of times action *a* has been visited from state *s; P(s,a)* is the prior possibility of selecting *a* at *s;* and c_{puct} is a constant that determines the degree of exploration.

3.4.2 Expansion and evaluation

During the expansion phase, the leaf node is expanded and evaluated by DNN to get the value *v* which is the prediction of final outcome, and the policy *p*, and the probability distribution of the best action. Then, each move is initialized to:

$$N(s_n, a_n) = 0$$

$$W(s_n, a_n) = 0$$

$$\mathrm{p}(\mathrm{s_n}, \mathrm{a_n}) = \mathrm{p_a}$$

Where $W(s_n, a_n)$ is the total values of all successor nodes that have been explored through the current node, and p_a is a randomly initialized weight from DNN.

3.4.3 Backup

After the expansion, the previous leaf node has now become a tree node. Starting from this node, MCTS retraces the nodes traversed to the root of the search tree and uses the updated information for each node:

$$N(s_n, a_n) = N(s_n, a_n) + 1$$

$$W(s_n, a_n) = W(s_n, a_n) + v_n$$

$$Q(s_n, a_n) = \frac{W(s_n, a_n)}{N(s_n, a_n)}$$

Where v_n indicates the value of the position. As the child nodes of the two adjacent layers in the search tree are different, the node values of the two adjacent layers will be opposite to each other.,

3.4.4 Play

By repeating the above three steps, MCTS can simulate self-playing. After all the simulations, it can basically cover most chess games and moves. How to play each move, what is the winning rate after playing, and what kind of situation can be found in the tree. Step 4 is to choose which move should be actually played in the current situation from the tree. The move to be really played at the root would be chosen in proportion to the visit count. Specifically, MCTS selects move a from root s0 by the following formula:

$$\pi(a \mid s) = \frac{N(s,a)^{1/\tau}}{\sum_b N(s,b)^{1/\tau}}$$

Where τ is the temperature parameter that controls the exploration level. At the end of self-playing, the MCTS's policy π and game outcomes (win, loss, or tie) would be saved as training data of the DNN.

4. Experiment and result

To compare the effectiveness of different tree search methods, we conduct several experiments. First of all, we carry out the experiment of the minimax program by increasing the search depth and playing chess with ordinary chess players. Then we record the winning time of the minimax algorithm for each round (10 games). In the second experiment, we record the time before each program makes a movement between minimax and alpha-beta pruning. In the third experiment, MCTS without an evaluation function play against alpha-beta pruning with the classical evaluation function, and the winner is recorded each time.

Table 1 shows the result of Chinese chess using minimax algorithm at different search depths. It can be seen that the deeper the search level is, the more evaluation information is obtained, and the higher the efficiency of the system is.

Table 1. Comparison of wining results

Search depth	Number of matches	Winning times
1	10	3
2	10	5
3	10	6
4	10	8

The performance comparison between minimax and alpha-beta pruning is shown in Table 2 as follows. It is found that alpha-beta pruning is significantly faster than minimax after more than 4 searching layers.

Table 2. Comparison of response time

Search depth	miniMax	Alpha-Beta pruning
1	<0.1	<0.1
2	<0.5	<0.1
3	5~8	<0.1
4	>100	<0.5
5	>300	1~3

Table 3 shows the results of different rounds of MCTS winning against alpha-beta pruning. We fix the depth of the Alpha-Beta pruning algorithm to 2 layers.

Table 3. Winner outcome list

	Winner
1	Alpha-Beta
2	Alpha-Beta
3	MCTS
4	MCTS
5	Alpha-Beta
6	MCTS
7	MCTS
8	Alpha-Beta
9	MCTS

It can be seen from the table that the performance of MCTS is only slightly better than alpha-beta pruning, which may be due to the unoptimized efficiency of the evaluation function of MCTS and the low number of matches between programs. We hope to obtain better experimental results by increasing the number of MCTS simulations and adopting optimized evaluation methods.

5. Conclusion

The research on computer chess games is still in its development stage. This paper introduces two popular game tree search algorithms commonly used in board games: the minimax algorithm and the alpha-beta pruning. The experimental results show that the optimal results of the search algorithm are very effective and practical when given a certain depth of information accumulation. The more layers it achieved, the better strategy it will make. When data mining technology is combined with tree search, the process of selecting the optimal node in the game tree can be improved. Besides, the Monte Carlo tree search and its improved AlphaZero method are discussed in this paper. It illustrates that the random moves based on the action selection probability parameters are helpful to improve the level of chess. At the same time, we propose to combine machine learning with deep neural networks and Monte Carlo algorithms to enhance chess ability that does not require professional chess knowledge. However, more testing is needed in this field in future work. The performance of the AlphaZero algorithm needs to be measured in environments with more complex conditions and greater memory requirements. At present, the implementation of such testing is limited by resources.

References

[1] Campbell, M., Hoane Jr, A. J., & Hsu, F. H. (2002). *Deep blue*. Artificial intelligence, 134(1-2), 57-83.

[2] Sang-Hun, C. (2016). *Google's computer program beats Lee Sedol in Go tournament*. New York Times.

[3] J. X. Chen, "*The Evolution of Computing: AlphaGo*," in Computing in Science & Engineering, vol. 18, no. 4, pp. 4-7, July-Aug. 2016, doi: 10.1109/MCSE.2016.74.

[4] Shannon, C. E. (1950). *XXII. Programming a computer for playing chess*. The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science, 41(314), 256-275.

[5] G. Wu, "*The Total Number Calculation of States of Chinese Chess Based on the Dynamic Programming of Computer Games*," 2021 33rd Chinese Control and Decision Conference (CCDC), 2021, pp. 2188-2191, doi: 10.1109/CCDC52312.2021.9601527.

[6] Strong, G. (2011). *The minimax algorithm*. Trinity College Dublin.

[7] Knuth, D. E., & Moore, R. W. (1975). *An analysis of alpha-beta pruning*. Artificial intelligence, 6(4), 293-326.

[8] Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., ... & Colton, S. (2012). *A survey of monte carlo tree search methods*. IEEE Transactions on Computational Intelligence and AI in games, 4(1), 1-43.

[9] Gelly, S., & Wang, Y. (2006, December). *Exploration exploitation in go: UCT for Monte-Carlo go*. In NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop.

[10] P. Auer, N. Cesa-Bianchi and P. Fischer, "*Finite-time analysis of the multiarmed bandit problem*", Mach. Learn., vol. 47, no. 2, pp. 235-256, 2002.

[11] Zhang, H., & Yu, T. (2020). AlphaZero. I*n Deep Reinforcement Learning* (pp. 391-415). Springer, Singapore.

[12] Liu, J., Qi, Y., Meng, Z. Y., & Fu, L. (2017). *Self-learning monte carlo method.* Physical Review B, 95(4), 041101.

[13] C. -H. Hsueh, K. Ikeda, S. -G. Nam and I. -C. Wu, "*Analyses of Tabular AlphaZero on NoGo*," 2020 International Conference on Technologies and Applications of Artificial Intelligence (TAAI), 2020, pp. 254-259, doi: 10.1109/TAAI51410.2020.00054.