Game Design of Gomoku AI

苏锦淼 09016406

Version 1.0

Algorithm ideas

First of all, let's assume a 15*15 game. Then, there are 255 possibilities for Black to take the first step. At this time, there are 254 moves for White. After choosing one of the 245 moves, the blacks have 253 may, and so on. If all the following methods are taken into account, they are listed as a huge game tree. The root node of the game tree is the first position of the black chess, followed by the second floor is all the possible situation of white chess at this time, and then down until the end of the game. The key to victory is to find the most favorable node in the lower sub-nodes and use the prediction to determine the best way to go. The more the computer predicts the number of steps, the better the situation of the chess game becomes, and the more likely the computer will win.

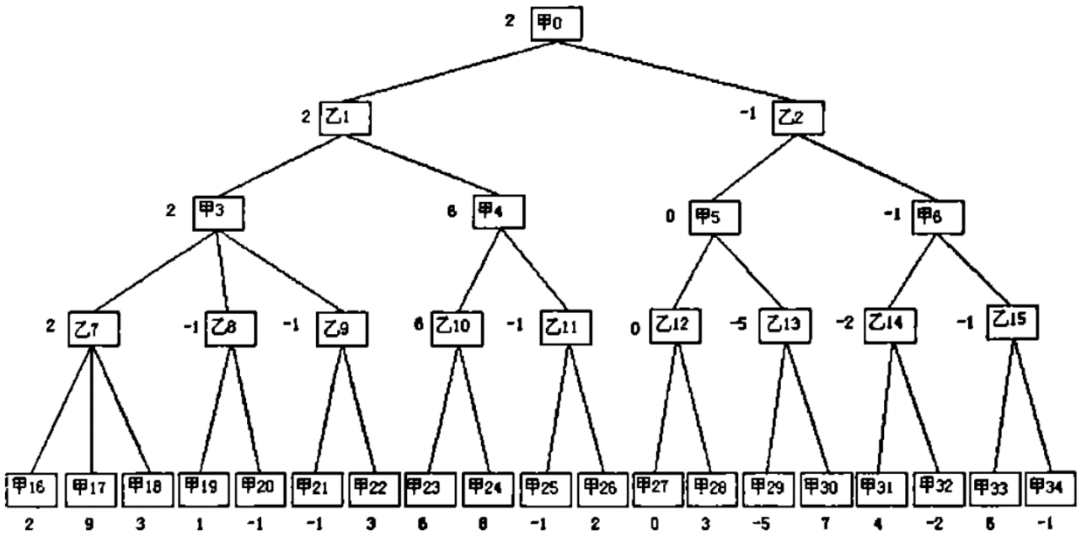
In choosing the best way to move, we need to use a minimax search algorithm, and the premise of implementing this algorithm is to have an evaluation function for the game. The following is to consider the specific implementation of this search function and evaluation function.

Search function

The search function part mainly uses the minimax search and alpha-beta pruning. In order to reduce the number of vacancies that need to be searched, it is tentatively stipulated that the point that can be fall after each step is a gap around that position.

Minimax Search

When falling a piece, the computer should consider the more favorable to the first step of his own, the more adverse to the opponent's next step, then the more beneficial to the second step of himself, and so on. Therefore, the maximum and minimum searches are represented by a minimax search tree. The function searches the tree layer by layer, selecting the maximum value for the own layer and the minimum value for the opponent until a certain depth.



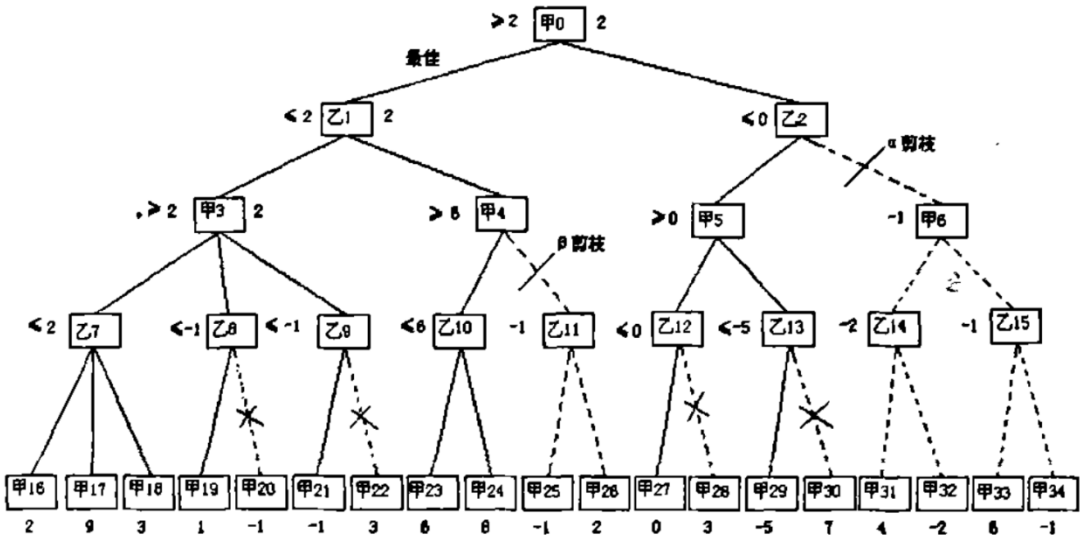
The above figure is the so-called game tree.

If: A corresponds to the computer side, B corresponds to the player. At present, the computer side plays chess(A0 node), the child node of the next layer is the player's next. Both sides hope that they will proceed in the direction that is most beneficial to one's own in the future situation. In order to get the high score, the computer will select the first child with the highest score of B1, whose score is 2. So, the score of A0 is 2. And B1 wants to have a low score, so he will select A3 with the lower score between A3 and A4. So, the value of B1 is 2. Therefore, in the

minimax search algorithm, the node A which desires maximum value is the maximum point, and the node B that wants the minimum value is the minimum point. The search process is a minimax search algorithm.

Alpha-Beta Pruning

The basic idea of the alpha-beta pruning algorithm is that the player does not make unfavorable choices for himself. According to this premise, if there is a bad situation in a child node that is obviously unfavorable to him, as a player who is smart enough, he will leave the worst case to his opponent. Therefore, this node does not need to continue to search, and it can be traced back. The so-called alpha and beta are the best worst-case scenarios for the player and your opponent.



As shown in the figure.

Alpha pruning: Taking the maximum point A3 and all the following nodes as an example. A3 is greater than or equal to 2 due to B7 is 2. A 19 is 1, then B 8 is less than or equal to 1. A3 is the largest. B can temporarily take 2. Thus, there is no

need to consider B8 which less than or equal to 1. Therefore, there is no need to consider A20. We can cut it off.

Beta pruning:

As shown in the figure, the minimum point B1 and all the following nodes can be analyzed in the same way.

The specific code is as follows.

```
int maxMinWithAlphaBetaCut(int chessBoard[MAXSIZE][MAXSIZE], int whiteOrBlack, int depth, Point opPos, int alpha, int beta)
{
    int bestValue, curValue;
    if (depth == 0)
    {
        bestValue = getBoardValue(chessBoard, 3-whiteOrBlack);
        return bestValue;
    }
    else
    {
        for (int i = 0; i <MAXSIZE; i++)
        {
            for (int j = 0; j <MAXSIZE; j++)
            {
                if (chessBoard[i][j] == 0)
                {
                    if (isIsolated(chessBoard, Point(i, j)))
                        continue;
                    chessBoard[i][j] = whiteOrBlack;
                    curValue = -maxMinWithAlphaBetaCut(chessBoard, 3 - whiteOrBlack, depth - 1, Point(i, j), -beta, -alpha);
                    chessBoard[i][j] = 0;
                    if (curValue >= beta)
                        return beta;
                    if (curValue > alpha)
                    {
                        alpha = curValue;
                    }
                }
            }
        }
    }
    return alpha;
}
```

Evaluation function

The evaluation function is to evaluate the current situation and return a score. At this time, the larger the score, the better the computer is. The smaller the score, the better the player. The evaluation function uses the difference between the scores of the two positions to measure the advantage. There are obviously the following value inequalities in Gobang:

Live three> Sleep three> Dead three

Live four> Washed four> Dead four

Live four> Live three> Washed four

Five pieces connected are biggest

Therefore, these are all taken into account reasonably when constructing the valuation function. These relationships are all reflected in the valuation function, achieving appropriate defense and offense.

The algorithm implementation of the valuation function is to calculate the weights on the situation. The more pieces are connected, the greater the weight of each piece which is connected. There are four directions on the board. They are horizontal, vertical, left diagonal, right diagonal. In each direction, the weights are calculated separately. There are four arrays, horizontalStatus[MAXSIZE][MAXSIZE], verticalStatus[MAXSIZE][MAXSIZE], rightStatus[MAXSIZE][MAXSIZE], leftStatus[MAXSIZE][MAXSIZE], storing the weight of the piece in each direction. Take the horizontal direction as an example. The computer holds white. When the first position is empty and white appears behind, the weight accumulates from 5. When the last position is still the white piece, the weight *5 will be the weight of the connected white pieces in the front. If it is not white, the weight will directly be the weight of the connected white game. When the first position is black and white appears behind, the weight accumulates from 1. When the last position is still the white piece, the weight *5 will be the weight of the connected white pieces in the front. If it is not white, the weight will directly be the weight of the connected white game. Table One behind can be an example.

The specific horizontal direction code is as follows.

```
for (int i = 0; i<MAXSIZE; i++)
{
    if (chessBoard[i][0] == whiteOrBlack)
        horizontalStatus[i][0] = 1;
    else
        horizontalStatus[i][0] = 0;
    for (int j = 1; j<MAXSIZE; j++)
        if (chessBoard[i][j - 1] == whiteOrBlack)
        {

            if (chessBoard[i][j] == whiteOrBlack)
                horizontalStatus[i][j] = horizontalStatus[i][j - 1] + 1;
            else
            {
                horizontalStatus[i][j] = 0;
                if (chessBoard[i][j] == 0)
                    horizontalStatus[i][j - 1] *= 5;
            }

        }
        else
        {
            if (chessBoard[i][j - 1] == 0)
            {
                if (chessBoard[i][j] == whiteOrBlack)
                    horizontalStatus[i][j] = 5;
                else
                    horizontalStatus[i][j] = 0;
            }
            else
            {
                if (chessBoard[i][j] == whiteOrBlack)
                    horizontalStatus[i][j] = 1;
                else
                    horizontalStatus[i][j] = 0;
            }

        }

}
for (int i = 0; i<MAXSIZE; i++)
{
    for (int j = MAXSIZE - 2; j >= 0; j--)
    {
        if (horizontalStatus[i][j] != 0 && horizontalStatus[i][j]<horizontalStatus[i][j + 1])
        {
            horizontalStatus[i][j] = horizontalStatus[i][j + 1];
        }
    }
}
```

Then the evaluation function is implemented as follows.

```
for (int i = 0; i<MAXSIZE; i++)
    for (int j = 0; j<MAXSIZE; j++)
    {
        value += calCube(horizontalStatus[i][j]) + calCube(verticalStatus[i][j]) + calCube(Status45[i][j]) + calCube(Status135[i][j]);
        if (horizontalStatus[i][j]==35||verticalStatus[i][j]==35||Status45[i][j]==35||Status135[i][j]==35)
            value=INF;
        if (horizontalStatus[i][j]==40||verticalStatus[i][j]==40||Status45[i][j]==40||Status135[i][j]==40)
            value=INF;
        if (horizontalStatus[i][j]==20||verticalStatus[i][j]==20||Status45[i][j]==20||Status135[i][j]==20)
            value=INF;
        if (horizontalStatus[i][j]==8||verticalStatus[i][j]==8||Status45[i][j]==8||Status135[i][j]==8)
            value=INF;

    }
```

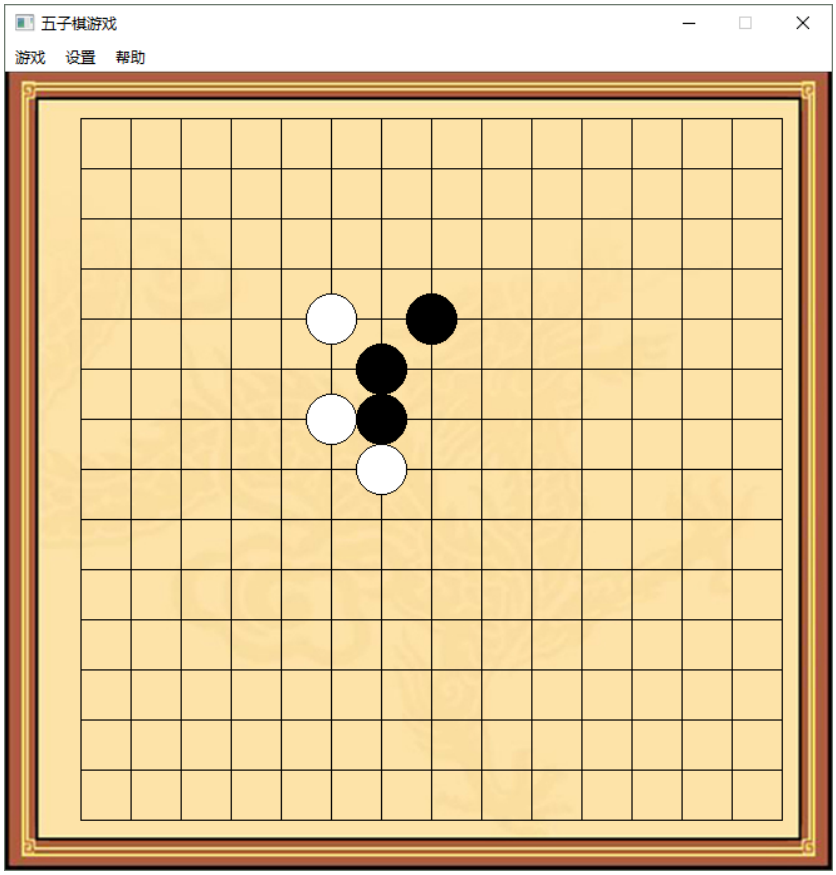
In particular, when (empty=0, white=1, black=2) 21111,11112,01111,01110, if it is a turn of the computer, the computer will win and set the value infinite. In this case, the

weight for the pieces are 20, 8, 40, and 35. Besides, only one of the four directions is required to satisfy the condition.

UI Design

In order to achieve a better visual interface design, I plan to use QT to complete this Gomoku game design, including the design of the board, menu, and sound effects. Predictably, the game mainly includes buttons such as "game", "settings", and "help". The user can directly click on the board to start the next Gomoku game, or set the "new game", "end game" and "quit game" through the "game" button. The player can also turn off or turn on the game sound effect through the "settings" button.

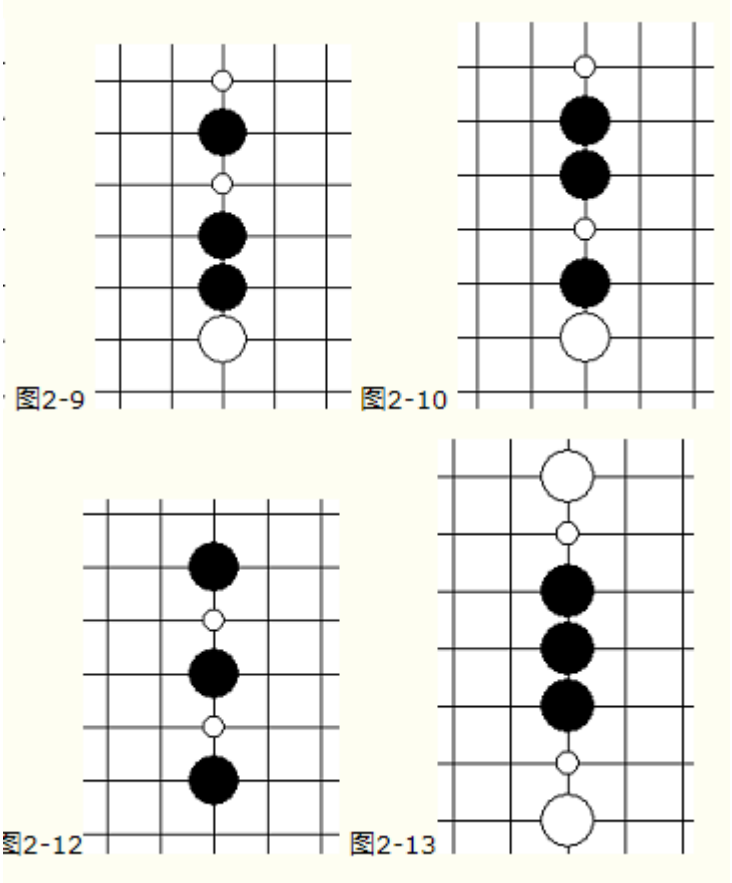
The target is as follows.



Algorithm Improvement Ideas

1. The valuation function is still not perfect enough. It cannot take some special types of chess into account.

2. The scope of the search function is limited and the range is slightly narrow. The range can be further expanded to achieve some special types of chess, making the AI more intelligent.



[some special types of chess]

3. The search function only relies on the alpha-beta pruning. So, the solution space is too large and the search depth is limited. Thus, the search function needs further improvement.

Table One

Chess type(empty=0,white=1,black=2)	Weight(every white piece)
01111	35
0112	6
2111	15
2112	2

Reference website

http://blog.csdn.net/ji_yun/article/details/46124201

http://zjh776.iteye.com/blog/1979748

https://www.cnblogs.com/songdechiu/p/5768999.html

http://blog.csdn.net/lihongxun945/article/details/50625267